79-0622

LEVEL II

SC

RATIONALE FOR THE DESIGN OF THE GREEN PROGRAMMING LANGUAGE

A language designed in accordance with the Steelman requirements

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway, Minneapolis, MN 55413

and

Cii Honeywell Bull
68 Route de Versailles
78430 Louveciennes, France

March 15, 1979

(6)

RATIONALE FOR THE DESIGN OF THE GREEN PROGRAMMING LANGUAGE

A language designed in accordance with the Steelman requirements.

(15) MDA903-77-C-0331
ARPA order-3341

Honeywell, Inc.
Systems and Research Center
2600 Ridgway Parkway, Minneapolis, MN 55413

and

Cii Honeywell Bull
68 Route de Versailles
78430 Louveciennes, France

(11) 15 March 15, 1979

(12) 264p.

Foreword

The Green language is the result of a collective effort to design a language satisfying the Steelman requirements. The design team was led by Jean D. Ichbiah and has included Bernd Krieg-Brueckner, Brian A. Wichmann, Henry F. Ledgard, Jean-Claude Heliard, Jean-Raymond Abrial, John G.P. Barnes, and Olivier Roubine. In addition, major contributions were provided by G. Ferran, I.C. Pyle, S.A Schuman, and S.C. Vestal.

Two parallel efforts started in the second phase of this design had a deep influence on the language. One was the design of a test translator, with the participation of K. Ripken, P. Boullier, J.F. Hueras, and R.G. Lange. The other was the development of a formal definition using denotational semantics, with the participation of V. Donzeau-Gouge, G. Kahn, and B. Lang. The entire effort benefitted from the dedicated support of Lyn Churchill and W.L. Heimerdinger.

At various stages of this project several persons had a constructive influence with their comments, criticisms and suggestions. They are P. Brinch Hansen, D.A. Fisher, G. Goos, C.A.R. Hoare, M. Woodger, and Mark Rain.

Over the two years spent on this project, three intense one-week design reviews were conducted with the participation of J.B. Goodenough, H. Harte, M. Kronental, K. Correll, R. Firth, A.N. Habermann, J. Teller, P. Wegner, and P. R. Wetherall.

These reviews, other comments by E. Boebert, P. Bonnard, T. Frogatt, H. Ganzinger, C. Hewitt, J.L. Mansion, F. Minel, E. Morel, J. Roehrich, A. Singer, D. Slosberg, and I.C. Wand; the numerous evaluation reports received on the preliminary design, the on-going work of the IFIP Working Group 2.4 on system implementation languages, and that of LTPL-E of Purdue Europe, all had a decisive influence on the final shape of the Green language.

Table of Contents

1. Introduction

This document, the Rationale for the design of the Green programming language, and the companion Reference Manual, are the two defining documents for the Green language. They serve different purposes.

The Reference Manual contains a complete and concise definition of the language. Following Wirth we believe in the virtue of having a rather short reference manual. This has the advantage of providing the information in a form that can easily be consulted, read and reread several times, as the basis for developing a good familiarity with the language.

Having a short reference manual however does not permit the infusion of much motivational information, and more generally, of information describing the reasons behind the major design decisions. Neither does it permit the insertion of the larger examples that are yet essential to help the reader get a better feeling for the language, and for the interaction among its features.

The Rationale is meant to serve precisely that purpose. It is divided in chapters covering different aspects of the language. Most chapters of the Rationale correspond to chapters of the Reference Manual. Expressions and statements are here regrouped in a single chapter, since the subject is fairly classical. Conversely, in view of the importance of the subjects, special chapters are devoted to numeric types and to access types, in addition to the chapter on types. All chapters of the Rationale are fairly independent (at the cost of some repetition) and can be read in any order after an initial pass over the Reference Manual.

Most chapters of the Rationale have a common structure. They start with an introduction to the topic discussed. An informal introduction to the language features follows. This informal introduction is made in terms of examples chosen to reflect the major classes of uses of the features considered. Some of these examples may be viewed as skeletons of actual systems.

We believe that the reader will get the spirit of the language reading these examples. They should help him develop an intuition for programming style in the Green language.

A discussion of the technical issues follows the informal introduction. Such discussions cover the major design decisions, their justification, and the interactions with other aspects of the language. In particular these discussions consider implementability: designing a programming language without a deep concern for implementability would be little more than an exercise in style. Accordingly, implementability has been an overriding concern in this design.

The main source of inspiration for the Green language is the programming language Pascal and its later derivatives. Pascal itself only meets a small part of the Steelman requirements. Merely to attempt to extend Pascal would have been neither feasible nor a desirable approach. Neither the simplicity nor the elegance that Pascal derives from its careful adaptation to its problem domain could be retained in such an approach. The extensions would inevitably interact with each other in undesirable ways. Hence, a goal in the design of the Green language was to retain the Pascal spirit of simplicity and elegance but not necessarily the form of each Pascal feature, since the problem domain defined by the Steelman requirement is much more ambitious.

How should the Green documents be read? Assuming a basic knowledge of Pascal, we recommend that the reader start with a quick first pass of the Reference Manual. After this pass, the various informal presentations included in the Rationale should provide a good basis for a second reading of the Reference Manual. The sections on technical issues should probably be read last.

2. Lexical and Textual Structure

A program is a text specifying actions to be performed by a computer. Programs are written by programmers and are processed by mechanical translators. The need to accommodate these two forms of communication is omnipresent at every level of consideration of a programming language, including the lowest levels where we are only concerned with the physical appearance of a program text.

The lexical and textual structure of a programming language is of course important for ease of program translation, and for translation time error detection. Its importance is even greater for readability (logical error detection) and teachability and these have been given major consideration in the design of the lexical and textual structure of the Green language. We believe that our understanding of programs can be greatly simplified if our intuition is able to rely on textual forms that convey the logical structure of the program. This justifies the special attention devoted to structural analogies.

2.1 Lexical Structure

A program is written in characters forming lines on the printed page. The arrangement on the page is primarily to assist the human reader, and consequently is mainly in a free format. Programs are required to have a representation in the ASCII character set, and may contain both upper case and lower case letters. In addition, every source program can be converted into an equivalent program that only uses a 55 character subset of ASCII.

On a higher level than that of characters, a program is considered to consist of lexical units. Both the machine and the human interpreter of programs will tend to work in lexical units, so it is important that such units should be clearly specified. Lexical units are clearly delineated and may not straddle line boundaries - a restriction that assists human reading and also compiler error recovery. The lexical units are:

- Identifiers, including reserved words and predefined attributes
- Single- and double-character delimiters
- Numbers (integer and real numbers)
- Character strings

In addition, a program text can include elements that have no influence on the meaning of a program but are intended for the human reader or for the compiler. These are

- Comments
- Pragmas

Identifiers start with a letter which may be followed by a sequence of letters and digits. In addition, an underscore may appear within an identifier. This underscore is significant and plays the role of the space in ordinary prose (but without breaking the integrity of the identifier). The need for such an underscore is seen from good choices of names such as BYTES_PER_WORD rather than BYTESPERWORD.

Reserved words are distinguished from program identifiers; there are 62 such words. Reserved words cannot be redeclared. Hence programmers cannot write obscure programs by redefining the meaning of such words. Similarly, programs can be highlighted by special printing of reserved words on an appropriate output device. The method chosen in this manual is boldface (and lower case). Since the language uses only one alphabetic font, one could envisage methods of highlighting the reserved words by the use of a different font, such as lower case, italics, underlining, etc. All these methods have the important property that the method of typing programs with the ASCII character set is not prejudiced. This is important, since it is currently possible to get excellent representations of programs via graphical printers of photocomposition machines.

Names for predefined attributes are from a distinct list, but are not reserved words since they are always preceded by a prime character and can thus be distinguished at the lexical level. The Green language uses predefined attributes to express predefined properties and environment enquiries. Other languages have used dot notation or functional notation for this purpose. Both of these forms have the disadvantage of restricting the user's choice of names.

For example if the address of an object were denoted by a function, this function would have to be overloaded on all data types; any user definition of ADDRESS would hide the predefined one and thus make it unavailable. Similarly, dot notation would prevent declaration of record components with the same identifier. Neither of these situations is acceptable in light of the fact that the number of predefined attributes can be large and that some of them may be particular to an implementation. All these problems are avoided with the Green notation.

The choice of identifiers for reserved words and predefined attributes depends primarily on convention. Preference is given to full English words rather than abbreviations. For instance, **procedure** is used rather than **proc** (in Algol 68) and **constant** rather than **const** (in Pascal). Shorter words were also given preference, for example **access** is used in preference to **reference**, and **task** is used in preference to **process**.

The following special characters can be used as delimiters between lexical units:

```
&  '  (  )  *  +  ,  -  .  /  :  ;  <  =  >  |
```

Further delimiters are constructed by juxtaposition of two (or three) such characters as follows:

```
=>  ..  **  :=  =:  :=:  /=  >=  <=  <<  >>
```

Naturally, where possible, printers may choose to print /= as ≠ and similarly >= as ≥ and <= as ≤.

Numeric literals are all introduced by an initial digit. A simple digit sequence is a decimal integer. For other bases from 2 up to 16 the base is given first in decimal form and is followed by a # sign (or the replacement character :) and by the sequence of *digits*. These may include the letters A to F for bases greater than ten. Thus, the conventional methods of setting bit patterns in binary, octal or hexadecimal are provided.

Real numbers must contain a decimal point or an exponent or both. The reasons for this are covered in the chapter on numerics (see 5). The underscore is permitted within a number to break up long sequences of digits, a requirement that has long been recognized by printers.

Another form of lexical unit is the string (and character literal). The characters are enclosed in quotes ", where two successive quotes represent the quote itself. Strings in the program are limited to a single line, to reduce the consequences of program errors due to omission of a quote character or ambiguity with respect to space characters. To specify a long string, the string is split across multiple lines and connected by the catenation operator. This is possible because catenation will be performed by the compiler. Automatic paragraphing processors may have to split a string in order to give a satisfactory layout. Apart from long strings, there may be a need to split strings that contain control characters or characters that are not in the 55 character subset of ASCII, to permit these to be represented. Examples of strings are as follows:

```
"A LONG LINE OF PRINTED OUTPUT WHICH" &
" IS CONTINUED ON THE NEXT LINE OF THE PROGRAM. "

"Strings start with the "" character."

"END OF LINE" & CR & LF & "START OF NEXT LINE"
```

Comments may appear alone on a line or at the end of a line. As an end of line remark, the comment should appear as an explanation of the preceding text -- hence the use of the double hyphen is appropriate, as illustrated by this sentence. For simplicity, no space is permitted between the two hyphens. No form of embedded comments (within a line of text) is provided as their use is too slight and not worth the extra complexity. For a further discussion of comments see [SW 74].

A pragma (from the Greek word meaning action) is used to direct the translation system in particular ways, but has no effect on the semantics of a program. Pragmas are used to control source text and listing, to define an object configuration (for example the size of memory), to control features of the code generated (particularly the level of diagnostics), etc. Such directives are not likely to be related to the rest of the language in an obvious way. Hence the form taken should not intrude upon the language, but it should be uniform.

The general form of pragmas is defined by the language. They start with the reserved word **pragma** and a pragma identifier optionally followed by a list of identifiers, strings, and numbers enclosed within parentheses. The pragma ends with a semicolon and may appear at any place where a declaration or statement may appear. Examples of pragmas are as follows:

```
pragma LIST(ON);
pragma OPTIMIZE (SPACE);
pragma INCLUDE ("common_options");
pragma SUPPRESS(OVERFLOW);
```

Some pragmas are defined by the language (see appendix B of the Reference Manual). It is expected that other pragmas will be defined as part of the support environment developed around the language.

2.2 Textual Structure

Above the lexical level, the text of a program has structure as an arrangement of lexical units. This structure is described by syntax in the conventional manner. However, a number of issues require separate exposition to clarify the decisions taken.

Declarations and statements are always terminated by semicolons. This departure from the Pascal practice, in which a semicolon is used as a separator, requires justification. Analyses of programmer errors support the use of the semicolon as a terminator [GH 75]. Also, inserting another declaration or statement is eased by this convention, since normal layout places the semicolon at the end of the line (requiring a change to two lines in the case of a separator).

The additional required semicolon also aids recovery by the compiler after a syntax error. On the other hand if semicolon is a terminator, recovery from a missing semicolon may be trivial for the parser.

Many program structures have simple brackets, for example the **loop** and **end loop** of the iterative statement, or **begin** and **end** of the subprogram body. Such structures can be indicated clearly by good program layout, which can be done automatically. Structures have been chosen to aid automatic program layout because the consistency of presentation is a significant advantage to the reader. It should be noted that conditional expressions (not present in Pascal or this language) pose severe problems for automatic layout.

Several textual program constructs exhibit a *comb-like* structure. This structure has simple bracketing and is best illustrated with the conditional statement, as in the following example:

```
if    A = 1  then
      S1;
elsif A = 2  then
      S2;
elsif A = 3  then
      S3;
end if;
```

The structure has three parts: an initial strong opening bracket (**if**), a set of weaker delimiters (**elsif**) and a strong closing bracket(**end if**).

The equivalent code can be written with the case comb structure:

```
case A of

   when 1 =>
      S1;
   when 2 =>
      S2;
   when 3 =>
      S3;
end case;
```

Note that an **else** S4 added to the conditional statement would correspond to the **others** construct in the case statement.

Further examples of the comb-structure are:

```
procedure P is
   ...
begin
   ...
exception
   ...
end P;
```

```
select
   ...
or when A =>
   ...
or when B  =>
   ...
else
   ...
end select;
```

It will be noted that each large-scale comb structure has the terminating reserved word **end**. For statements and declarations this is followed by an initial reserved word as well:

if	**end if**
loop	**end loop**
case	**end case**
select	**end select**
record	**end record**

For named program units such as subprograms and modules (and also *accept* statements since they act as bodies for the corresponding entries) the name of the unit may follow the final **end** to assist recognition of the structure.

In general, language constructs which do not express similar ideas should not look similar. Thus, unlike Pascal, which uses a colon in both cases, the Green language uses different notations for statement labels (for goto statements) and for choices (in case statements). Statement labels have angle brackets << >> placed around the identifier of the label. They emphasize that this is a special point in the program. Conversely, choices express preconditions for executing the statements which follow. Choices thus are indicated in clauses of the form

```
when ... =>
```

The same structure has also been used in select statements since it corresponds to the same idea of a precondition.

Whenever possible a uniform notation is used for similar constructs. The clearest example of this is the *case* structure which is used for variant records as well as statement selections. Although in Pascal variant records and case statements are similar constructs their notation is not uniform. This causes difficulty in teaching the language and is a source of errors. The contrast is best illustrated by an example:

A graphics data structure in Pascal

```
type FIGTYPE = (POINT, CIRCLE, TRIANGLE);
type FIGURE =
  record
    X, Y: REAL;
    case F: FIGTYPE of
      POINT:;
      CIRCLE: (RADIUS: REAL);
      TRIANGLE:
        (SIDE: array [1 .. 2] of
               record
                  ANGLE, LENGTH: REAL;
               end
        )
end
```

The equivalent structure in the Green language:

```
type FIGURE is
  record
    X,Y  : REAL;
    F    : constant (POINT, CIRCLE, TRIANGLE);
    case F of
      when POINT     => null;
      when CIRCLE    => RADIUS : REAL;
      when TRIANGLE =>
        SIDE: array (1 .. 2) of
              record
                 ANGLE, LENGTH: REAL;
              end record;
    end case;
  end record;
```

A case statement in Pascal

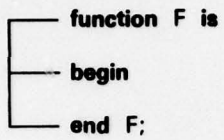
```
case DAY of
  MON : STARTWORK;
  FRI   : TIDYUP;
  TUE,WED,THU : WORK;
  SAT, SUN:
end
```

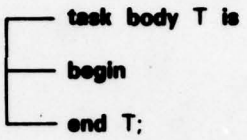
The equivalent case statement in the Green language

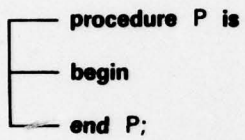
```
case DAY of
  when MON => START_WORK;
  when FRI   => TIDY_UP;
  when TUE .. THU   => WORK;
  when SAT | SUN   => null;
end case;
```

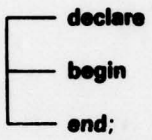
The structural analogy in the Green language between the declarative case (variant part) and the case statement should assist the human reader and help in learning the language. The approach will also simplify pretty printing of programs by mechanical tools which do not necessarily have access to declarative information.

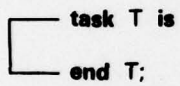
A similar analogy exists between the case and the select statement. Other structural analogies, which are adequately reflected by the syntax, correspond to the textual structure of functions, procedures, tasks, packages and blocks.

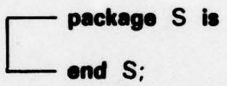












These structural analogies have been used quite systematically. They should develop a feeling of familiarity for the reader and simplify the teaching of the language.

3. Expressions and Statements

Programs achieve actions by executing statements. These may contain expressions that are formulas defining the computation of values. Expressions (including their constituent constants, aggregates, and variables) and statements are fairly *classical* aspects of most programming languages. Hence we limit the discussion to the most prominent points.

3.1 Variables and Constants

Although the approach to variables follows Pascal, there are some differences. First, initialization expressions can be inserted at the point of declaration. This avoids long initialization sequences that are divorced from the declarative code and hence hard to locate. Second, references to array components use round rather than square brackets. The advantage of this is a uniform notation within an expression for reference to array components and for function calls. The same argument of *uniform referents* [Ro 70] justifies using the same syntax for component selection of records whether they are statically or dynamically allocated (see Chapter 6).

A delicate balance is necessary in the handling of constants and variables. Reliability and security demand that the two concepts be clearly separated. On the other hand it is convenient if a program can be changed by simply altering the declaration of an identifier from a variable to a constant.

The Green language meets these goals by having two distinct but related forms of declarations for the two concepts. Thus

```
C : constant INTEGER := 300_000;
```

is a constant declaration, whereas the following declaration defines a variable with an initial value.

```
SPEED : INTEGER := C/1500;
```

The handling of constants in Pascal is rather inflexible. All constants must appear before the other declarations [WSH 77] and no translation time evaluation is allowed.

The Green language allows constant valued expressions wherever a constant is permitted. Furthermore, it follows Algol 68 in permitting constants whose values are determinable at scope entry time. The **constant** qualifier which appears in the declaration of an identifier means that the value of the corresponding object cannot be altered after the initialization. Hence the translator shall forbid any attempt to alter its value. However it does not mean that this value must necessarily be known at translation time. Constant declarations are like other declarations; they may be mixed in groupings that reflect the logical needs of a program.

3.2 Aggregates

Aggregates denote values of array and record types. In both cases the Green language admits a positional notation in which the aggregate is given by a sequence of values, implicitly associated with the corresponding components. Another notation that is often preferable is also provided, in which the association of given values to the components is defined by explicit selectors called choices. This discussion concentrates on array aggregates.

An array aggregate defines an array value. As for literals of other types, the question of the type of an array aggregate has to be solved. By analogy consider the assignment

```
I := 8 + 5;
```

Assuming that the variable I is of type MY_INTEGER, we are able to identify a "+" operation that takes two arguments of type MY_INTEGER and delivers a result of type MY_INTEGER. Hence we may interpret the literals 8 and 5 as being of type MY_INTEGER and perform the corresponding operation, provided that these literals satisfy the range constraint of the type MY_INTEGER.

The same logic applies to an assignment involving an array aggregate. An array aggregate is an array value of a certain array type which must be identified from the context. This means that we need to identify the type and constraint of the array value:

- the component type
- the number of dimensions
- the index type for each dimension
- the bounds of each dimension

As in the case of integer literals, we must be able to deduce all these elements from the context, otherwise the aggregate is ambiguous. Consider for example an array A defined by

```
type VECTOR is array (NATURAL) of INTEGER;
A : VECTOR(1 .. 10);
```

We first examine non-positional aggregates and then positional aggregates.

Non-Positional Aggregates

Suppose that we have

```
A := (1 .. 6 => 0,  7 .. 10 => 100);
```

The aggregate must describe a value of type VECTOR, that is, a one dimensional array of integers. In addition, its indices must be natural numbers in the range 1 .. 10.

Having derived this information from the analysis of

```
A :=
```

we can now interpret all choices in the above aggregate as legitimate index values, since 1, 6, 7, and 10 are indices in the range 1 .. 10. Conversely, we would be able to detect that the following aggregate is illegal:

```
A := (11 .. 16 => 0,  17 .. 20 => 100);    -- illegal!
```

since the index values 11, 16, 17, and 20 are not in the range 1 .. 10.

Another possible viewpoint for non-positional array aggregates would be to consider them as values independently of the context and then to invoke some *sliding* semantics for assignments and comparisons. Such a semantics would permit

```
A := (101 .. 110 => 0);    -- illegal in Green since A'FIRST = 1, A'LAST = 10
```

Such an aggregate would be interpreted as a value of an array AGGREGATE declared as

```
AGGREGATE : array (101 .. 110) of INTEGER := (101 .. 110 => 0);
```

Then the assignment could be interpreted as a slice assignment

```
A(1 .. 10) := AGGREGATE (101 .. 110);
```

There are several difficulties with this alternative semantics of named aggregates. First, it would make the determination of the bounds of an aggregate quite difficult in general, if not impossible, as in

```
(15|16 => 0,  6 .. 8 => 1,  others => 5)
```

Second, this interpretation would not be possible for index types defined by enumeration. Consider for example

```
type SCHEDULE is array (DAY'FIRST .. DAY'LAST) of BOOLEAN;
```

and an array aggregate such as

```
(WED .. SUN => TRUE,  others => FALSE)
```

This aggregate can only mean that we have TRUE for the days WED to SUN and false for MON and TUE. No sliding semantics could give this interpretation.

For these reasons the semantics retained in the Green language is that the index values given in each component association must be index values for an array identified from the context.

Positional aggregates

For positional aggregates we must take a different viewpoint since there are no explicit selectors indicating the destination of each value. Conceptually a positional aggregate must thus be viewed as a sequence of values of the component type. Note that this same viewpoint is taken when we deal with slice assignments. Hence

```
A := (9, 8, 7, 6, 5, 4, 3, 2, 1, 0);
```

can be interpreted as the sequence of assignments

```
A(1)  := 9;
...
A(10) := 0;
```

This is to be compared with

```
A := B(11 .. 20);
```

which is interpreted as the sequence of assignments

```
A(1)  := B(11);
 ...
A(10) := B(20);
```

3.3 Expressions

The purpose of an expression is the computation of a value. The evaluation of an expression is conceptually an indivisible operation. Hence it must be assumed that all of the constituent expressions will be evaluated and that any exception condition that can arise in any part of the expression cannot be avoided. In this sense

```
A = 0 or X/A > 10
```

is not a valid expression, since the validity of the second member depends on the value of the first member. Whenever there is such a conditionality it must be made explicit in a control structure, perhaps with short circuit conditions (see 3.7) in order to emphasize the possibility of incomplete evaluation.

The language provides six operator precedence levels, listed below in increasing order of binding strength.

(*lowest*)	logical	**and**	**or**	**xor**				
	relational	=	/=	<	<=	>	>=	
	adding	+	-	&				
	unary	+	-	**not**				
	multiplying	*	/	**mod**				
(*highest*)	exponentiating	**						

We found this number of levels to be the minimum number compatible with accepted practice. Clearly different levels are required for relational, adding, multiplying and exponentiation operators (unary does not really constitute a level). In addition, logical operators must be on another lower level if expressions such as

```
A = B  or  C = D
```

are to be written without parentheses. In the case of a succession of operators with the same precedence the Green language adopts the traditional rule of left to right evaluation. In addition the syntax requires explicit parentheses in the case of a succession of different logical operators. For example

```
A or (B xor C)      -- parentheses required
(A or B) xor C      -- parentheses required (not always equal to previous expression)
A or (B and C)      -- parentheses required
A or B or C         -- no need for parentheses
A and B and C       -- no need for parentheses
```

In practice most expressions are simple; the inconvenience of parentheses should not be exaggerated as programmers tend to introduce additional parentheses for readability purposes (to reassure themselves when in doubt) whether they are required or not.

3.4 Statements

The classical forms for sequential processing are included in the Green language: assignment statements, subprogram calls, conditional, iterative and transfer statements. Less classical control structures such as the raise statement and the statements used for parallel processing are discussed in later chapters.

It is convenient to distinguish normal statements and transfer statements. For a normal statement, execution continues with the following statement. For transfer statements (*exit*, *return*, *goto*) control will transfer to another place in the program. Another useful distinction is between simple and compound statements. A simple statement contains no other statement. Compound statements may include other sequences of statements.

The language follows the rule that wherever a statement may appear, a sequence of statements may appear. This rule simplifies program modification since insertion of a statement can be done without the need to insert extra **begin** and **end** delimiters as is the case in Algol and Pascal. Statements of a sequence of statements are executed in sequence unless a transfer statement is encountered.

Although it is redundant, an explicit *null* statement has been included in the language for readability. For instance if nothing is required for a given alternative of a case statement, it is preferable to state this explicitly with the statement

```
null;
```

rather than convey the same impression by an empty statement that could be taken as an unintentional omission.

3.5 Assignment Statements

The assignment statement is generally regarded as the simplest of all statements. In fact, this is the case only if precautions are taken in its definition. Since calls to value-returning procedures can only appear in an expression on the right hand side of an assignment (and since no other side effects are possible within expressions), the meaning of an assignment statement does not depend on whether its left or right side is evaluated first. Hence an assignment may be viewed by a program reader as an indivisible action. The same argument of simplicity justifies Pascal in not providing either multiple assignment or parallel assignments. In any case, the multiple assignment of Algol 60 is rarely used except for initialization and also causes parsing problems for the human reader.

Some additional precautions are required for slice (i.e. subarray) assignments. If assignment is to be permitted between slices it should follow the same axiomatic rules as for ordinary variable assignments. If this were not the case it would be questionable to use the same syntax for two operations which have different logic. For example, after the assignments

```
C  := B;
A  := B;
```

we can deduce from the axiomatics of assignment that A = C. This same consequence will apply to slices only if overlap is forbidden. Consider for instance the (incorrect) assignments

```
C (1 .. 5) := B(2 .. 6);
B (1 .. 5) := B(2 .. 6);    -- illegal assignment
```

Because of the overlap, we do not have C(1 .. 5) = B(1 .. 5) but rather C(1) = B(2), C(2) = B(3), C(3) = B(4), C(4) = B(5). Similarly the incorrect assignments

```
C (2 .. 6) := B(1 .. 5);
B (2 .. 6) := B(1 .. 5);        -- illegal assignment
```

would fail to produce C(2 .. 6) = B(2 .. 6) and the actual results would even depend on whether the second assignment is considered to be performed on an element-by-element basis or whether an intermediate copy is used.

In conclusion, overlap in slice assignment is a form of aliasing that is hence forbidden, and causes the exception OVERLAP_ERROR. Note that overlapping slice assignments have sometimes been used in low-level programming to achieve an effect similar to what can be (legally) performed with an aggregate assignment such as

```
A(1 .. 80) := (1 .. 80 => " ");
```

3.6 If Statements

If statements are used to select statement lists for execution on the basis of a condition. The syntax

```
if condition then
   sequence_of_statements
end if;
```

is fairly classical. The **if** and **end if** bracket structure avoids the dangling else ambiguity. If statements containing **elsif** clauses can be used to select alternative statement lists depending on different conditions:

```
if RAIN then
   -- sequence of statements describing
   -- what to do when it rains
elsif SUN_SHINE then
   -- sequence of statements describing
   -- what to do when the sun shines
else
   -- sequence of statements describing
   -- what to do for other weather conditions
end if;
```

Strictly speaking, **elsif** clauses are redundant: the corresponding statements can always be rewritten in the form of nested if statements. However this nesting is generally awkward and does not convey the correct impression, namely that the alternatives are on the same level, apart from the fact that the conditions should be evaluated in the order in which they appear.

3.7 Short Circuit Conditions

Boolean expressions, like other expressions, may be rearranged by the translator as permitted by the properties of their operators. Thus commutativity can be used in expressions such as A **and** B. Depending on the complexity of the term B, it may be more efficient (on some but not all machines) to evaluate B only when the term A is true. This however is an optimization decision taken by the compiler. In any case the expression A **and** B should always deliver the same result as B **and** A, and should have no other effect.

In some situations, however, we may want to express a conjunction of conditions where each condition should be evaluated (has meaning) only if the previous condition is satisfied. This may be done with short circuit conditions such as:

```
if I /= 0  and then  A/I > B then
   ...
end if;
```

Clearly it would not be legal to express this condition as a boolean expression, since an exception would result if I were zero and the second term were evaluated first. Similarly, short circuit disjunctions can be expressed with **or else** clauses as in the following example:

```
exit when X = null  or else  X.AGE = 0;
```

In this case the condition following **or else** will only be evaluated if the previous condition is not satisfied. Since the rules of evaluation of **and then** clauses and **or else** clauses are thus contradictory, the two forms of clauses cannot be mixed in the same condition.

In Algol 60 one can achieve the effect of short circuit evaluation only by use of conditional expressions, since complete evaluation is performed otherwise. This leads to cumbersome constructs which are awkward to follow:

```
if(if I /= 0 then true else A div I > B) then...
```

Several languages, including Pascal, do not define how boolean conditions are to be evaluated. In fact the Pascal CDC 6000 series compiler uses complete evaluation, whereas the ICL 1900 compiler uses short circuit, i.e., treats **and** as **and then**, and similarly **or** as **or else**. As a consequence programs based on short circuit evaluation will not be transferable. This clearly illustrates the need for separating boolean operators from short circuit conditions.

3.8 Case Statements

A case statement serves to execute one element of a list of alternatives selected on the basis of the value of an expression. Each possible alternative is preceded by the list of values for which the corresponding alternative should be selected. The main criteria in the design of the case statement have been reliability and generality.

For reliability, the translator must be given the possibility to check for accidental omission of some alternatives. For that reason it is required that all possible values of the type of the discriminating expression be provided for in the selections. Naturally, a qualified expression can be used to restrict this choice and the selection **others** may be used to represent all values not specified. As an example consider the declarations

```
type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
subtype WORK_DAY is DAY range MON  .. FRI;
subtype REST_DAY  is DAY range SAT  .. SUN;

D : DAY;
```

With the above declarations all values of the type DAY (the type of D) must appear in one selection, as in

```
case D of
  when MON | TUE | WED | THU | FRI => WORK;
  when SAT  | SUN => REST;
end case;
```

This could have been written in the equivalent form

```
case D of
  when MON | TUE | WED | THU | FRI => WORK;
  when others => REST;
end case;
```

If it is known in a given context that the case discriminant belongs to a given subtype, a case with a qualified expression may be used. Only the values of the corresponding subtype can appear in the selections. Should the constraint not be satisfied (for example if D = SAT), an exception RANGE_ERROR would result.

```
case WORK_DAY(D) of
  when MON | WED | FRI => LATE;
  when TUE  | THU => EARLY;
end case;
```

The second concern in the design of case statements is generality: the syntax of selections should accommodate all situations which are likely to arise, given that the case discriminant has a discrete type. Hence it should permit lists of values as well as ranges. Thus the first example above is more likely to be written using subtype names:

```
case D of
  when WORK_DAY'FIRST .. WORK_DAY'LAST  => WORK;
  when REST_DAY'FIRST  .. REST_DAY'LAST  => REST;
end case;
```

or using the ranges directly:

```
case D of
  when MON .. FRI   => WORK;
  when SAT  .. SUN  => REST;
end case;
```

Such ranges are very useful in case selections. They avoid long lists that can be tedious to read and error prone.

A very important diagnostic facility that the translator should provide is the listing of all values of the discriminant type that do not appear in any listed choice. For an erroneously incomplete case statement the translator has the information and should provide it for the programmer. For enumeration types with a large number of values, should this kind of diagnostic not be provided it might be quite difficult for the programmer to discover any missing values.

```
for I in 1 .. 10 loop
   ...
end loop;

for J in reverse A .. B loop
   ...
end loop;
```

In both cases, unlike Pascal, the *loop parameter* is considered as local to the loop. It is implicitly declared by its appearance in the *for* iteration condition. Its value is constant for each iteration.

More complicated forms of loop constructs such as Zahn's construct [Za 74] and the related construct provided in Modula [Wi 76] have been considered in this design but in the end rejected. As shown in the example below, situations for which such constructs would be used can be written quite easily with the existing forms.

```
declare
   ESCAPE : (A, B, NORMAL) := NORMAL;
begin
   for ... loop
      ...
      ESCAPE := A;  exit;
      ...
      ESCAPE := B;  exit;
      ...
      ESCAPE := A;  exit;
      ...
   end loop;

   case ESCAPE of
      when A   => ...
      when B   => ...
      when NORMAL => ...
   end case;
end;
```

The major emphasis in the design of the loop primitives has been on simplicity: loops should have an intuitive meaning and users should not have to consult a reference manual to understand their meaning. Several studies on the use of programming languages have shown that the vast majority of loops are very simple. Hence generalities such as the *step* expression of Algol 60 should be avoided. The redundancy provided for conditional exits is motivated by textual considerations: loop termination conditions should be marked very conspicuously. Hence

```
exit when CONDITION;
```

is certainly more explicit than the equivalent form:

```
if CONDITION then
   exit;
end if;
```

Guarded commands were also considered in this analysis and not retained. They have advantages for the development of program proofs. However, they are not compatible with other looping constructs with explicit exits. Hence if they were retained it would have been to the exclusion of other loop forms, a decision which seemed too drastic.

```
for I in 1 .. 10 loop
   ...
end loop;

for J in reverse A .. B loop
   ...
end loop;
```

In both cases, unlike Pascal, the *loop parameter* is considered as local to the loop. It is implicitly declared by its appearance in the *for* iteration condition. Its value is constant for each iteration.

More complicated forms of loop constructs such as Zahn's construct [Za 74] and the related construct provided in Modula [Wi 76] have been considered in this design but in the end rejected. As shown in the example below, situations for which such constructs would be used can be written quite easily with the existing forms.

```
declare
   ESCAPE : (A, B, NORMAL) := NORMAL;
begin
   for ... loop
      ...
      ESCAPE := A;  exit;
      ...
      ESCAPE := B;  exit;
      ...
      ESCAPE := A;  exit;
      ...
   end loop;

   case ESCAPE of
      when A   => ...
      when B   => ...
      when NORMAL => ...
   end case;
end;
```

The major emphasis in the design of the loop primitives has been on simplicity: loops should have an intuitive meaning and users should not have to consult a reference manual to understand their meaning. Several studies on the use of programming languages have shown that the vast majority of loops are very simple. Hence generalities such as the *step* expression of Algol 60 should be avoided. The redundancy provided for conditional exits is motivated by textual considerations: loop termination conditions should be marked very conspicuously. Hence

```
exit when CONDITION;
```

is certainly more explicit than the equivalent form:

```
if CONDITION then
   exit;
end if;
```

Guarded commands were also considered in this analysis and not retained. They have advantages for the development of program proofs. However, they are not compatible with other looping constructs with explicit exits. Hence if they were retained it would have been to the exclusion of other loop forms, a decision which seemed too drastic.

Case statements are conventionally implemented with an implicit transfer table. This table will generally contain one entry for each possible value of the discriminant type. Quite often however, if some of the alternatives perform no actions, the translator may optimize the code generated by using a shorter table and an explicit range check. As an example

```
case D of
   when SAT    => SHOP;
   when SUN    => SLEEP;
   when others => null;
end case;
```

may be compiled to produce a code equivalent to

```
if D in REST_DAY then
   case REST_DAY(D) of
      when SAT    => SHOP;
      when SUN    => SLEEP;
   end case;
end if;
```

thus leading to a two entry transfer table. Finally, case statements with very sparse selections or with ranges as selections can be compiled as equivalent if statements. Thus for our first example we may have:

```
if D in  WORK_DAY then
   WORK;
else
   REST;
end if;
```

3.9 Loop Statements

The main form of loop statement, called a basic loop, allows conditional or unconditional exit statements to appear anywhere within the statement list of the loop:

```
loop
   READ_CHARACTER(C);
   exit when C = "*";
   PRINT_CHARACTER(C);
end loop;
```

Although this form of loop is quite general, a special form also exists to single out the cases in which a continuation condition appears at the start of the loop:

```
while MORE_TO_DO loop
   ...
end loop;
```

Similarly two forms of *for* loops are provided to iterate over ranges either in normal (increasing) or in reverse (decreasing) order (the range bounds must be in increasing order):

4. Types

4.1 Introduction

The notion of type has gradually emerged from the past twenty years of the history of programming languages as the way by which we impose structure on data. A now widely accepted view of types is that a type characterizes the set of values that objects of the type may assume, and the set of operations that may be performed on them. This common view is also taken in the Green language definition.

There are several important reasons for which it is found desirable to associate a type with constants, variables, and parameters of subprograms:

- *Factorization of Properties, Maintainability*

  Knowledge about common properties of objects should be collected in one place and given a name. A type declaration serves that purpose. Subsequently, this type name may be used to refer to these common properties in object declarations. Furthermore, maintainability is enhanced since a change of properties has only to be effected at a single point of the program text, the type declaration.

- *Abstraction, Hiding of Implementation Details*

  Abstract or external properties of objects and operations should be separated from underlying and internal implementation dependent properties such as the physical representation on a specific machine. Only the abstract properties of an object need to be known for its use. Implementation details should be hidden from the user. The need for such a separation is particularly strong in the case of disjoint sections of a program text, produced and maintained by different programmers, and possibly separately compiled.

- *Reliability*

  Objects with distinct properties should be clearly distinguished in a program and the distinction should be enforced by the translator. Requiring that all objects be typed thus contributes to program reliability. Experience has shown that a well written program in Pascal can be recognized easily by the use made of the typing facility to increase the reliability, readability and security of the program.

Several classical problems are associated with the formulation of a type facility in programming languages. Some are a subject of ongoing debate among language designers and users, in particular:

(a) Static versus Dynamic Properties

Should both the static properties (those which are determinable from an analysis of the program text at translation time) and the dynamic properties (those which may depend on the dynamic behavior of a program, such as reading from an input device) be covered by a single notion of type?

(b) Type Equivalence

Should the language provide some form of *equivalence* or *compatibility* among types with logically related properties?

(c) Parameterization

Should the language provide some form of parameterization for types and their associated properties? Should the evaluation of type parameters be performed entirely at translation time or be deferred until execution time?

The Green solutions to the above problems are now summarized. A detailed discussion of design decisions is given in later sections.

(a) Static versus Dynamic Properties

Two notions are distinguished: the notion of type and the notion of subtype. A type characterizes a distinct set of values and its static properties, such as the applicable operations. Constraints may be imposed on named types, for example a range constraint for a scalar type or an index constraint for an array type. In general, these constraints cannot always be determined at translation time.

A subtype name serves as an abbreviation for a type name and a constraint associated with the type. Several difficulties in the types of Pascal noted by Habermann and others [Hab 73, WSH 77] are overcome in the Green language by the notion of subtype.

(b) Type Equivalence

Each type definition introduces a distinct type. In consequence, each type name denotes a distinct type. Objects with distinct types cannot be intermixed.

In contrast, objects belonging to different subtypes of the same type are compatible. They may be assigned to variables with differing subtypes of the same type. Constraints are checked during translation whenever possible and during execution otherwise.

Types defined as derived from another type are said to be conformable with it. Explicit conversions are possible between conformable types.

(c) Parameterization

Type definitions and their associated operations can be encapsulated in modules. A module can be parameterized by a *generic* clause with the consequence that all contained definitions, including those of types, are parameterized. Generic modules must be instantiated at translation time. Each instantiation must be explicit; it supplies values for the parameters and yields new types.

Parameterization at execution time is closely associated with the notion of constraint. In particular this applies to array and record types:

- An array type definition can leave index bounds unspecified. These are subsequently specified by an index constraint for a given array object so that different array objects of the same type may have different numbers of components. If such an array is a formal parameter of a subprogram, its bounds are obtained from the actual parameter for each call.

- A record type may have variants, i.e. alternative definitions of its components. Different variants are associated with the values of a special component called a discriminant, and it is possible to constrain a record to a single variant by use of a discriminant constraint. Variants are otherwise discriminated at execution time.

These solutions are detailed in the following sections. We first introduce the notion of type definitions and the resulting rule for determining if two objects have the same type. Constraints and subtypes are then discussed. Finally we discuss derived type definitions.

The specific properties of numeric types are discussed in Chapter 5, those of access types in Chapter 6. Parameterization with generic clauses is discussed in Chapter 13.

4.2 Type definitions

The language provides the capability to define new types. The language construct used to introduce a new type is called a type definition. Examples of type definitions appear below.

```
range -2**24 .. 2**24                    -- integer type definition
(LOW, MEDIUM, HIGH)                      -- enumeration type definition
digits 8 range 0.0 .. 1E24               -- floating point type definition
delta 0.01 range 0.0 .. 1_000.0          -- fixed point type definition

array (1 .. 128) of CHARACTER            -- array type definition
record RE, IM : INTEGER; end record      -- record type definition
new INTEGER                              -- derived type definition
access STRING                            -- access type definition
```

As stated before, one of the objectives of a type system is reliability. It should prevent erroneous mixing of objects of different types. Hence a key issue in the design of a type system is the formulation of the conditions that must be satisfied by two variables (or constants) in order that they should have the same type.

Alternative solutions for the issue of type equivalence have been formulated in a paper by Welsh et al. [WSH 77]. These solutions are classified into two families called *name* equivalence and *structural* equivalence.

The solution used in the Green language is related to name equivalence. It is based on the principle that every type definition introduces a distinct type. Two type definitions introduce two distinct types even if they are textually identical. For example, the objects A and B declared by

```
A : (ON, OFF);
B : (ON, OFF);
```

belong to two distinct types since they are declared in terms of two distinct type definitions. On the other hand the objects C and D declared by

```
C, D : (LOW, MEDIUM, HIGH);
```

belong to the same type since they are declared in terms of a single type definition.

Type definitions given directly in object declarations as in the above examples are allowed and are called *anonymous* type definitions. They are sometimes useful for defining the type of some record components.

In the majority of cases, however, a type definition is given a name in the type declaration where it appears. A consequence of the principle stated above is that two distinct type names always refer to two distinct types. Thereafter two objects have the same type if they refer to the same type name in their declaration. The expression *name equivalence* derives from the fact that this is by far the most usual way for two objects to have the same type. As an example consider

```
type PERSON is
   record
      SEX : constant(MALE, FEMALE);
      case SEX of
         when MALE    =>  ENLISTED   : BOOLEAN;
         when FEMALE  =>  PREGNANT   : BOOLEAN;
      end case;
   end record;
X  : PERSON;
Y  : PERSON;
```

The variables X and Y both belong to the same type since they both refer to the same type name (PERSON) in their declaration (note that the selected components X.SEX and Y.SEX belong to the same anonymous type).

Structural equivalence refers to solutions where some form of equivalence rule is formulated between types on the basis of their properties. We have rejected structural equivalence in order to avoid pattern matching problems for the translator and for the human reader. We also believe that structural equivalence tends to defeat the purpose of strong typing since objects may be considered as being of the same type because their structures are identical by accident, or because they have become identical as a result of textual modifications performed during program maintenance. Such objects can then be mixed erroneously without causing translator diagnostics.

Further arguments in favor of name equivalence are presented in later sections.

4.2.1 Scalar and Access Types

Scalar types cover enumeration types, integer and real types.

Enumeration types are defined by enumeration of their values. These are considered to be in increasing order. The same literal may appear in more than one enumeration type. This is a logical consequence of the fact that character sets can be defined as enumeration types. Tt would not be acceptable for example to require the character "A" to appear exclusively in one character set. Enumeration literals that may denote enumeration values of different types are said to be *overloaded.* As an example consider

```
type COLOR  is (RED, ORANGE, YELLOW, GREEN, BLUE, VIOLET);
type LIGHT  is (RED, AMBER, GREEN);
```

As the enumeration value GREEN appears in two types, it may be necessary to qualify GREEN by the desired type in some contexts, for example COLOR(GREEN) or LIGHT(GREEN).

Structural equivalence of enumeration types is undesirable since it may involve comparisons of long identifier lists. Furthermore, the usual maintenance problem would exist, since two enumeration types could accidentally become equivalent by *inserting or deleting an element.*

The maintenance argument is similar for numeric types. Two real types should not be equivalent just because their error bound specifications happen to be the same. The programmer should be encouraged not to use just FLOAT or LONG_FLOAT as the type of the variables, but to introduce the particular precisions required for the applications and to express the commonality by a type declaration.

An access type definition introduces a distinct collection of dynamic objects. The space for that collection can be released once the scope of the access type definition is left, since different collections are associated with different access types. Using structural equivalence for access types would introduce a difficulty since two structurally equivalent access types could appear in different scopes.

4.2.2 Record types

To emphasize the arguments against structural type equivalence rules, we next discuss record types. Consider, for example:

```
type COMPLEX is
  record
    RE  : INTEGER;
    IM  : INTEGER;
  end record;

type RATIONAL is
  record
    NUMERATOR     : INTEGER;
    DENOMINATOR   : INTEGER range 1 .. INTEGER'LAST;
  end record;
```

Several alternative forms of structural equivalence rules can be considered, involving increasing amounts of checking, especially if the record types have a large number of components:

(a) two types are equivalent if the texts in the right-hand side of their declarations (after *is*) are identical (disregarding textual layout such as space or new line characters, etc.);

(b) two types are equivalent if their component names and the type names of their components agree, in the same order;

(c) same as (b) but the names of the components need not agree, only their order. This is a more mathematical point of view, where one considers a record as a cartesian product;

(d) same as (b) but the order of components is not significant;

(e) same as (b) but only the type names of the corresponding components must be the same, not the names of the components nor the constraints;

(f) same as (e) but the subtypes must be equivalent;

(g) same as (e) but the component types must be equivalent, while their names need not be identical;

(h) same as (g) but a type name is also equivalent to the (possibly anonymous) text of the right-hand side of its declaration.

The types COMPLEX and RATIONAL given above would be equivalent under all the rules if their component names were accidentally the same and if the constraint on DENOMINATOR were not expressed in the type declaration. More specifically, under rule (b), COMPLEX would be equivalent to

```
type ANOTHER_COMPLEX is
  record
    RE, IM : INTEGER;
  end record;
```

Rule (c) makes sense for a language with positional notation only. It complicates the checking by the compiler, since all permutations must be considered. Conversely, rule (d) is sensible for a totally non-positional language where component names must always be specified for record aggregates. Rule (e) complicates the implementation of constraints and subtypes for components, since they must be checked for each component on record or array assignments. Rule (f) cannot be checked statically. Rule (g) requires a recursive matching algorithm. In addition, rule (h) requires type expansion and even an algorithm of cycle reduction in the case of mutually recursive access types.

All these complexities for the implementation and, above all, for the user, are avoided in Green by adopting the simple rule that every type definition introduces a distinct type.

4.2.3 Array Types

Arguments similar to those against the structural equivalence of record types hold for array types. It is certainly desirable to distinguish between arrays whose component type is different; the insistence that even their component subtypes be the same is well motivated by efficiency considerations. Consider, for example:

```
type LINE      is array (1 .. 128) of CHARACTER;
type TEXT_LINE is array (1 .. 128) of CHARACTER range "A" .. "Z";
```

Assignment of an array of type LINE to an array of type TEXT_LINE requires an implicit loop to check the range constraint for each component individually. An explicit conversion is required, which informs the reader of this potential cost (the absence of implicit conversion follows from the general rules of the language; the possibility of an explicit conversion follows from the rule explained in section 4.5 below).

One should also distinguish between arrays with distinct index types, even if their component type is the same:

```
type OPTION_SET is array (OPTION'FIRST .. OPTION'LAST) of BOOLEAN;
type COLOR_SET  is array (COLOR'FIRST .. COLOR'LAST)   of BOOLEAN;
```

An array of type OPTION_SET should not be compatible with one of type COLOR_SET just because the number of options happens to be the same as the number of colors. From a conceptual point of view, the two array types have nothing to do with each other, apart from their common component type.

On the other hand, it is reasonable to establish a relationship between arrays that differ only in their index ranges, as long as their index types and component types are the same. As shown in section 4.3 below, this relationship can be expressed by using array subtypes.

To summarize, the rule that two type definitions always introduce two distinct types is applied uniformly to all type definitions. As a consequence two distinct type names always refer to two distinct types. Commonality of properties is expressed in general by using the same type name.

4.3 Constraints and Subtypes

As mentioned before, a type characterizes a set of values that objects of the type may assume and a set of operations applicable to those values. Membership of an object in a type is a static property resulting directly from the declaration of the object.

It is possible to restrict the set of allowed values of a type without changing the set of applicable operations. Such a restriction is called a *constraint*, and the subset of values it defines is called a *subtype*. Membership of an object in a subtype can result from the object declarations. However, this need not always be the case since it is possible to define several overlapping subtypes (named or not) of a given type. Membership in a subtype may or may not be determined statically.

4.3.1 Constraints

A constraint can be used to restrict the set of allowable values of a type as in the following example:

```
CHARACTER range "A" .. "Z"
```

Such constraints may effectively be used by the compiler for optimization purposes. Their major use, however, is for greater security in assignment. Violations are reported at translation time when possible, or at execution time by raising an appropriate exception.

Consider the following example:

```
declare
   subtype LETTER      is CHARACTER range "A" .. "Z";
   subtype HEX_LETTER  is LETTER    range "A" .. "F";
   subtype EFG_LETTER  is LETTER    range "E" .. "G";

   A : CHARACTER  := "X";
   B : LETTER     := "Y";
   C : HEX_LETTER := "F";
   D : EFG_LETTER := "E";
begin
   A := B;    -- no check necessary
   B := C;    -- no check necessary
   B := A;    -- check that (A in LETTER), i.e.
              -- check that (LETTER'FIRST <= A) and (A <= LETTER'LAST)
   C := D;    -- check that (D <= HEX_LETTER'LAST)
   D := C;    -- check that (EFG_LETTER'FIRST <= C)
end;
```

4.3.2 Subtypes

Knowledge about the sharing of special properties should be factored. For this purpose, a subset of elements of a given type can be named in a subtype declaration, as in

```
subtype LETTER is CHARACTER range "A" .. "Z";
```

A subtype serves as an abbreviation for a type name and a constraint (more precisely, for the result of evaluating the constraint at the point of the subtype declaration). Since subtype declarations do not introduce new types, objects of different subtypes of the same type are compatible for assignment. Such objects can be mixed in expressions as long as the constraints are obeyed, as noted in the examples above.

Again, any change to a constraint in an explicitly named subtype declaration only requires a single text change. Note that subtype incompatibilities are checked, for example the subtype declaration

```
subtype FIFTEEN_FIRST is LETTER range "A" .. "0";    -- incorrect!
```

The character "0" (zero), given by mistake instead of the letter "O", does not belong to the subtype LETTER.

4.3.3 Evaluation of Constraints

Constraints may involve expressions that cannot be evaluated statically. The compiler can do little or no optimization in these cases. Constraints that determine critical space representations must therefore be known at translation time. Confinement to static evaluation would be much too restrictive in general. The assertions involved in range constraints would be too coarse, ranges could not be used as general loop iterators, and arrays could only be of static size.

An issue to be considered is the time at which the expressions appearing in constraints should be evaluated. Consider the subtype declaration:

```
subtype S is T range U .. V;
```

where U and V may be arbitrary expressions. The rule adopted in the Green language definition is that such expressions are evaluated when the subtype declaration is elaborated. This means that the subtype declaration is equivalent to the following sequence

```
U_0  : constant T := U;
V_0  : constant T := V;
subtype S is T range U_0 .. V_0;
```

where *U_0* and *V_0* represent identifiers not used elsewhere. Note that if the bounds of the range are not known at translation time, a descriptor (containing the values of *U_0* and *V_0*) must be implicitly generated by the compiler. Hence, for efficiency reasons, it is important to isolate the knowledge about equivalent constraints into one subtype declaration and to use the name of this subtype instead of repeating the constraint in several variable declarations. The values of the implicit constants *U_0* and *V_0* are denoted by the subtype attributes S'FIRST and S'LAST.

Note also that, for reliability and maintainability, using a subtype such as S is far better than repeating the corresponding constraint (**range** U .. V) at various points of the text, since the values of U and V at these points might differ. Thus it is preferable to write

```
declare
   subtype S is T range U .. V;
   X  : S;
   A  : array (S'FIRST .. S'LAST) of T;
   ...
   procedure P(Z : S) is ... end P;
begin
   for I in S'FIRST .. S'LAST loop
      if A(I) in S'FIRST .. S'LAST then
         ...
      end if;
   end loop;
end;
```

rather than to repeat the corresponding range in terms of the defining expressions U and V at various points of the text.

The rule that constraints are evaluated when the declaration where they are given is elaborated, is applied uniformly in all cases. In particular it applies to constraints of subprogram parameters. Making an exception for this case would increase the complexity of the language and also of the implementation since a subtype description would have to be maintained for every subprogram call (not just one for the subprogram declaration) because of the possibility of recursion. For these reasons the simpler, uniform rule has been retained.

4.3.4 Record Variants

A record type with a variant part specifies several alternative variants of the type. This means that the set of possible record values is the union of the sets of values possible for the alternative variants. Seen in this light, a variant of a record type is a subtype of the record type.

A variant part depends on a special component of the record, called its *discriminant*. Each variant defines the components that exist for a specific value of the discriminant. This is thus a form of parameterization of record types. However, unlike the parameterization that can be achieved by the generic facility, this parameterization can be dynamic: it can depend on the value of the discriminant and this value need not be known statically (at translation time).

When declaring a record variable it is possible to specify that its discriminant must always have a given value. This specification is a discriminant constraint that restricts the set of possible record values to those of the designated variant. The translator may take advantage of this knowledge when setting the amount of space reserved for the record variable. We may also define a subtype of a record type by associating the discriminant constraint with the record type name. We illustrate these possibilities with the following example:

```
declare
   type PERSON is
   record
      BIRTH  : DATE;
      SEX    : constant(M,F);  -- discriminant
      case SEX of
         when M  => ENLISTED   : BOOLEAN;
         when F  => PREGNANT   : BOOLEAN;
      end case;
   end record;

   subtype MALE   is PERSON(SEX => M);
   subtype FEMALE is PERSON(SEX => F);

   ANYONE : PERSON;
   HE     : MALE;          -- Equivalent methods of
   PETER  : PERSON(M);     -- declaring males
   JOAN   : FEMALE;
   SHE    : FEMALE;
begin
   ...
   ANYONE := HE;                 -- Valid, no checking required since MALE
                                 -- is a subtype of PERSON

   ANYONE := JOAN;               -- Similarly no check needed

   HE := PETER;                  -- No check needed, both are males
   HE := JOAN;                   -- Translation time error since MALE and FEMALE
                                 -- are disjoint subtypes of PERSON

   SHE := ANYONE;                -- Execution time check necessary; it will
                                 -- raise an exception if ANYONE is not a female

   SHE := FEMALE(ANYONE);        -- Equivalent to above, but useful redundancy
                                 -- to emphasize the possible exception condition
end;
```

When accessing a component of a constrained record such as JOAN or PETER, the value of its discriminant and hence the associated variant is already known. The constraint is part of the static information known about such record, and any assignment to the record variable is checked with respect to the constraint either at translation or at execution time. Consequently, references to record components such as JOAN.PREGNANT or PETER.ENLISTED are perfectly secure.

When accessing a component of a record that is declared without a constraint such as ANYONE, more precautions should be taken. A part of this security derives from the fact that discriminants are deferred constants and are not directly assignable. They may be set when the record value is defined and may only be changed by assignment to the record as a whole. Thus

```
ANYONE := PERSON(BIRTH => 1940, SEX => M, ENLISTED => TRUE);
```

is a legal complete record assignment which sets SEX equal to M. Similarly, assignments such as ANYONE := JOAN; or ANYONE := PETER; are complete record assignments which consequently set the value of the discriminant. However we must consider such a complete record assignment as defining a new object and the value of the object's discriminant cannot be changed separately. Thus an assignment such as

```
ANYONE.SEX := F;  -- illegal!
```

is forbidden and will be rejected by the translator.

The second key element to the security of variants is that access to a component of a variant is only legal if the discriminant has the corresponding value. This means that an access to the component

```
... ANYONE.ENLISTED ...
```

is equivalent to the following text.

```
if ANYONE.SEX /= M then
   raise DISCRIMINANT_ERROR;
end if;
 ... ANYONE.ENLISTED ...
```

Naturally the compiler can omit this implicit discriminant check in contexts where explicit checks are made, or when the explicit constraints make such checks unnecessary. Such explicit discrimination may take several forms. It can be achieved by an assertion or a renaming declaration specifying a subtype. It can also be achieved by an if statement

```
if ANYONE.SEX = M then
   -- access to ANYONE.ENLISTED requires no implicit check
end if;
```

or similarly by a case statement

```
case ANYONE.SEX of
   when M  =>
      -- access to ANYONE.ENLISTED requires no check
   when F  =>
      -- access to ANYONE.PREGNANT requires no check
end case;
```

Of course, the check can only be omitted as long as the discriminant is not changed as a result of a complete record assignment. Consider for example:

```
case ANYONE.SEX of
  when M =>
     ... ANYONE.ENLISTED ...     -- occurrence 1
     ... ANYONE.ENLISTED ...     -- occurrence 2
     UPDATE(ANYONE);
     ... ANYONE.ENLISTED ...     -- occurrence 3
     PRINT(ANYONE);
     ... ANYONE.ENLISTED ...     -- occurrence 4
  when F=> ...
end case;
```

No checks are needed for the first two occurrences. A check is needed for the third (assuming the mode of the parameter of UPDATE to be **in out**) but no check is needed for the fourth occurrence (assuming the mode of the parameter of PRINT to be **in**).

Note that additional problems exist if a record is shared by two tasks. One task could perform a complete record assignment (thereby changing the discriminant) while another is reading a component. We consider this problem to be a danger inherent in the use of shared variables rather than a problem concerning the formulation of record types. The tasking facilities of the language are powerful enough to make such shared variables virtually useless. If they are nevertheless used, the appropriate precautions should be taken by the programmer. On the other hand, we did not believe it correct to disfigure the semantics of the language because of such possible misuse.

It might be felt that the checking code is a high price to pay. This is, however, essential for security with variant records. Previous experience with languages such as Simula, which force a similar discrimination of variants, show that these checks are not as frequent as one might suppose. The parts of the programs that operate on a given variant tend to be textually discriminated as well as dynamically discriminated. Hence the checks can be achieved at a rather low cost (see also [We 78]).

One should not underestimate the importance of secure access to components of a variant part. A recent experiment [Ha 77] with a Pascal compiler in which this facility was offered as an extension reported that these checks caught a quarter of the initial errors in a large program.

4.3.5 Array Types with Unspecified Bounds

An array type declaration must specify the type of the array components and the type of each index, but it need not specify the actual bounds of each index. This means that the set of possible array values defined by the type contains arrays with different numbers of components. Consider for example:

```
subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;

type STRING is array(NATURAL) of CHARACTER;
```

Values of the type STRING are arrays of components of type CHARACTER indexed by natural numbers. However different string values need not have the same index bounds and hence the same number of characters.

It is possible to partition the set of array values into subsets corresponding to some fixed index bounds. Each such subset defines a subtype of the array type. The constraint used to fix the range of values of a given index is called an index constraint. For example

```
BUFFER : STRING(1 .. 1000);
```

defines an array object of type STRING whose index lower and upper bounds are the natural numbers 1 and 1000. As usual we can also define a subtype by naming the association of the type name with the index constraint:

```
subtype LINE is STRING(1 .. 120);

HEAD_LINE  : LINE;
BLANK_LINE : constant LINE := (LINE'FIRST .. LINE'LAST => " ");
```

Note that the other form of array type declaration (where the bounds are directly specified) is in fact a special case where the index constraint is directly given with the array type declaration. Thus the type declaration

```
type SCHEDULE is array (DAY'FIRST .. DAY'LAST) of BOOLEAN;
```

can be viewed as a contraction of

```
type schedule is array (DAY) of BOOLEAN;                  -- arbitrary number of days
subtype SCHEDULE is schedule(DAY'FIRST .. DAY'LAST);      -- always 7 days
```

For any object of an array type, the bounds of each index must be known. We have seen that they can be specified by an index constraint; they can also be obtained from the initial value:

```
TITLE   : STRING := HEAD_LINE;
MESSAGE : constant STRING := "HOW MANY CHARACTERS?";
```

Similarly, for a formal parameter they can be obtained from those of the corresponding actual parameter. Thus a subprogram with a parameter of such a type applies to all arrays of the type independently of their index bounds (unless of course an index constraint is given for the parameter declaration). For example concatenation for a string of arbitrary length is defined as

```
function "&"(X,Y : STRING) return STRING is
   RESULT : STRING(1 .. X'LENGTH + Y'LENGTH);
begin
   RESULT(1 .. X'LENGTH) := X;
   RESULT(X'LENGTH + 1 .. RESULT'LAST) := Y;
   return RESULT;
end "&";
```

This last example shows that array types with unspecified bounds can be viewed as a form of type parameterization. Again, unlike the parameterization that can be achieved by generic clauses (which is purely at translation time), this parameterization can be dynamic. For example, the bounds of the strings can vary from call to call. Such array types do not introduce new implementation problems since descriptors for index constraints are needed in any case whenever the bounds of an array are not known at translation time.

4.4 Derived Type Definitions

The basic mechanisms for introducing a new type are by enumeration and by composition from existing ones, in type definitions. Certain basic characteristics are automatically acquired by such a type, for example the availability of literals, aggregates, and operations such as indexing, component selection, etc. Another way of introducing a type is by means of a private type declaration (see 8.3.3).

A third possibility is provided in the Green language. A type B is declared as *deriving* its characteristics from those of another existing type A if it is declared as

```
type B is new A;
```

This form of type declaration is very useful whenever a type B is to have the same characteristics as a type A, and possibly some additional ones. All operations available for objects of type A at the point of declaration of B are inherited by B, although A and B nevertheless remain distinct types. Conversions between the two types are possible but they must be explicit, and must use the type name as qualification.

The relationship of *derivation* established by derived type declarations is transitive but not symmetric. For example in

```
type C  is new B;
type D  is new A;
```

the type C implicitly derives all characteristics of A through B. Similarly D derives all its characteristics from those of A but no direct explicit conversion is possible between C and D. Conversions must be achieved indirectly step by step, via the common ancestor A. The relation can thus be depicted as a strict hierarchy.

For example, consider

```
type STRING is array (NATURAL) of CHARACTER;

type LINE    is new STRING(1 .. 160);

type CARD  is new STRING(1 .. 80);

type CONTROL_CARD   is new CARD;

type PROGRAM_CARD  is new CARD;

type DATA_CARD        is new CARD;
```

Objects of type LINE cannot be accidentally mixed with those of type CARD, However, they can both be converted to objects of type STRING (of appropriate length) by means of *explicit* conversions. Similarly, CONTROL_CARD, PROGRAM_CARD, and DATA_CARD are distinct descendants of CARD.

Derived type definitions are useful for the definition of numeric types, for example when a user introduces a type such as

```
type MY_REAL is new FLOAT;
```

This type inherits all operations that are so far available for the type FLOAT. If for example a function SQRT defined as

```
function SQRT(X : FLOAT) return FLOAT;
```

is available, the effect of the derived type definition is to produce a function equivalent to the one defined by

```
function SQRT(X : MY_FLOAT) return MY_FLOAT;
```

Note that if it were not possible to inherit such library functions conveniently, the user might be tempted to work with the predefined types. This would endanger portability since the latter are implementation dependent.

If a constraint is given in the derived type definition, it does not affect the operations and values that are inherited but must be interpreted as applying to all objects of the declared type. Hence the declaration

```
type MY_INT is new INTEGER range 1 .. 1000;
```

can be viewed as equivalent to

```
type     my_int is new INTEGER;
subtype  MY_INT is my_int range 1 .. 1000;
```

One consequence of this definition is that for I of type MY_INT the relation

```
I < 2000
```

is legal since the value 2000 is inherited from the type INTEGER, and the relation is of course TRUE since I must be in the range 1 .. 1000.

Floating point and integer type definitions are interpreted in terms of derived type definitions. Thus a floating point type definition such as

```
type MY_REAL is digits 10;
```

is mapped by the compiler as an appropriate predefined floating point type (say FLOAT in this case) with sufficient precision. Hence it is equivalent to the following declaration

```
type MY_REAL is new FLOAT digits 10;
```

4.5 Explicit Conversions Between Array Types

The idea of name equivalence (rather than structural equivalence) has been used systematically in this design; in particular it has been used for array type definitions. Remember that, aside from simplicity, one of the main arguments in favor of name equivalence is the need to avoid accidental type equivalence. On the other hand this argument does not apply for explicit conversions; being explicit they are unequivocally intentional and cannot be accidental.

Explicit type conversions are clearly desirable among array types satisfying certain conditions. To illustrate their need consider first a package defining sorting operations. It could appear as

```
package SORTING is
   type VECTOR is array (INTEGER) of REAL;
   procedure SORT(X : in out VECTOR);
   ...
end SORTING;
```

Similarly a package for performing listing of arrays could be specified as

```
package LISTING is
   type TABLE is array (INTEGER) of REAL;
   procedure LIST(X : in TABLE);
   ...
end LISTING;
```

These two packages are of general use and hence they would probably be made available as library packages. A user trying to perform both sort operations (the operations of the type VECTOR) and listing operations (the operations of the type TABLE) would be faced with a contradiction if explicit type conversions were not available since any declared array can only be of one type. In addition a user may have declared an array as

```
A : array (1 .. 1000) of REAL;
```

without knowing in advance that he would ever sort (or list) such arrays. It would be undesirable to have to modify the declaration of A because the array needs to be sorted in one part of this program.

For these reasons explicit conversions are permitted between two array types (named or not) if the index types for each dimension are the same (or derived from each other) and if the component types are the same (or derived from each other). Such explicit conversions are expressed as qualified expressions. Thus our example above can appear as

```
declare
   use SORTING, LISTING;
   A : array (1 .. 1000) of REAL;
begin
   ...
   SORT (VECTOR(A));
   ...
   LIST(TABLE(A));
   ...
end;
```

Note that conversions are also possible when the constraints on the component type are different. Consider for example the array types

```
type CHAR_LINE is array (1 .. 120) of CHARACTER;
type TEXT_LINE  is array (1 .. 120) of CHARACTER range "A" .. "Z";
CL : CHAR_LINE;
TL : TEXT_LINE;
```

Explicit conversions such as

```
CL := CHAR_LINE (TL);
TL := TEXT_LINE (CL);
```

are allowed. The fact that they are explicit warns the user that they may (but need not) be costly. For *in out parameters implemented by reference*, such conversions may require the creation of a copy on the calling side if the compiler has chosen different representations for the two types.

4.6 Conclusion on Type Parameterization

In the design of the Green language we have very carefully introduced a distinction between parameterization at translation time and at execution time, especially in the case of types.

Parameterization at translation time is achieved by the *generic facility (see Chapter 13)*. The basis for this facility is quite traditional. It is context dependent macro substitution. Hence it can be used very freely, and it can be implemented efficiently.

For parameterization at execution time this design has chosen to remain extremely conservative, and to provide this only in domains in which it is already traditional, such as variant records and array types.

On the other hand we consider the problems of complete parameterization at execution time to be a subject of current research (see [HW 79][BJ 78]). To the best of our knowledge, solutions to this problem are not available, unless one is willing to abandon all efficiency considerations and interpret type parameters at execution time. Others may yet show that solutions do exist with efficient implementation, and that we were too cautious.

5. Numeric Types

5.1 Introduction

From the earliest days of computing, numerical calculations have played a dominant role in the use of computers. The need for a method of representing numbers that would rapidly handle a wide range of values, even if the representation were approximate, resulted in floating point hardware in the second generation of machines. Despite the long history of numeric computation, the majority of programming languages have obvious defects in their handling of both fixed point and floating point data types.

Fortran is very widely used for scientific computation and compilers are available on almost all machines. Several large high-quality packages have been implemented in Fortran and made available on a wide range of computers. Examples are the Numerical Algorithms Group (NAG) library of subroutines, the computer graphics package GINO-F and the engineering design system GENYSIS.

Nevertheless, numerous defects in the language can easily trap the unwary. Implicit truncation on assignment to an integer is an obvious trap that is compounded by the lack of any definition in the standard of the semantics of the truncation [ANSI 66]. No facilities are provided for fixed point variables, although there is a significant (but small) need for them, especially on computers without floating point hardware.

5.1.1 Floating Point: The Problems

Surprisingly, control of floating point precision is the most difficult area, and no completely adequate solution is available. Fortran does not define the accuracy of single precision, which in consequence varies on common systems from 24 to 48 bits in the mantissa. Therefore it is necessary to choose between single and double precision according to the implementation being used.

Changing precision is extremely awkward: declarations can easily be altered although implicit declarations will not be changed, floating point literals must have their exponents changed, all intrinsic functions altered, etc. Some functions have no double length counterpart (there is no DFLOAT, for instance) and hence careful checks are necessary. The Numerical Algorithms Group overcame this by an elaborate text processing package (which can handle other problems as well)[HF 76]. Programming conventions can be used so that the change is possible with a simple text edit, but no simple solution is available; for instance, use of double length throughout is not effective because of its excessive cost, nor is changing the type by IMPLICIT because it is not standard Fortran and literals cannot be changed in this way.

Several languages in the Algol 60 tradition (Pascal, Coral 66, RTL/2, etc) admit only one floating point data type. In some cases, such a simple solution can meet the users requirements better than can Fortran. The two Algol 60 compilers for the IBM 360 allow the user to determine (by a compiler directive) whether 32 or 64 bit precision will be used - a substantially easier task than the corresponding change in Fortran. In essence, there is only one floating point concept, and hence, unless the declarations can determine the precision, it is best to have only one data type.

Precision control in Algol 68 is by declaration of **long real** or even **long long real**, but since the accuracy is implementation dependent, declarative changes to the program are still necessary. Nevertheless, very simple textual changes to an Algol 68 program could map floating point declarations into an efficient program giving the required accuracy. Any language with user-defined types and *some* method of precision control provides the essential mechanism for an effective solution. Careful use of the typing facility to permit simple remapping is, of course, imperative.

5.1.2 Fixed Point: The Problems

There is also considerable difficulty in formulating a satisfactory fixed point facility. The Steelman requirements specify an exact representation and operations for exact computation. Part of this requirement was met with the exceedingly simple facilities in the preliminary definition of the Green language. Extensions merely to provide multiplication and division give problems. Applications demand arbitrary scales, not just powers of 2 or 10, in which case the scale of a product may not match the required scale. It is possible to use the greatest common divisor algorithm to determine the scaling required, but this is incompatible with a simple semantics that is so important for programmer comprehension.

Cobol apparently meets the Steelman requirements, but only by using decimal scales. Decimal scales are not adequate for two reasons: first, this is not necessarily the scaling required by the application, and second, 10 is too coarse for the standard 16-bit minicomputer. A glance at a Cobol manual will also indicate that explaining the implicit decimal point to the programmer is not as easy as it may seem.

One possibility is to allow the user to specify a number of scales connected by fixed conversions, such as seconds and minutes. Although this meets some application needs well, it is insufficient in other cases. The extensions required seem endless.

It is important to note that the package module facility of Green allows the user to write his own library for exact computation using integers. Compared with built-in facilities for fixed point, the main difference is the lack of a good notation for literals and some loss of efficiency. Since language simplicity is such an important goal, fixed point should only be included if the need is clearly established and the requirement well specified. Neither position can be confirmed, and hence we conclude that exact fixed point facilities should not be included in the language design.

An analysis of actual applications in many real-time situations reveals that there is a need for cheap floating point, as has been noted in many reports to the US Department of Defense. Small but frequently executed computations are performed upon digital input signals. The very simplest machines may not have floating point and hence some means is required to perform the computations quickly (i.e. software or firmware emulation of floating point is not fast enough). To say that in the future floating point hardware will always be available may not be the answer. First, floating point data require more space and second, source data input is initially fixed. Hence approximate fixed point is a better match to common application needs than floating point.

As we shall see, it must be admitted that programming with fixed point is much more difficult than with floating point. On the other hand, fixed point is potentially more reliable because tight bounds must be placed upon data values in order to perform effective error analysis.

Our conclusion is that approximate fixed point is the most likely avenue to provide an arithmetic capability complementary to integers and floating point.

5.1.3 Overview of Numerics in Green

The facility for numerics is based upon the idea that a numeric variable has an abstract value. Such values can be thought of as subsets of the real numbers. Exact computation is only provided with the integers. Approximate computation is provided in two forms: fixed point with an *absolute* bound on the error, and floating point with a *relative* bound on the error. The approximate types are called real types since they can be thought of as approximations to the mathematical concept of a real number.

The semantics of each numeric operation is derived from the type of its operands. Since the language provides overloading of operators, the operations available to the programmer include not only those derived from the predefined types, but also those introduced by the programmer.

The facility for numerics is based upon three unnamed types, called *universal_integer*, *universal_float*, and *universal_fixed*. These types are abstractions of the specific types in a given implementation.

(1) Universal Integer

There is an implementation defined type INTEGER which is equivalent to the type *universal_integer* with

```
range INTEGER'FIRST .. INTEGER'LAST
```

as the range constraint. This constraint reflects the maximum integer range (often one word) of the machine. Integer constraints are of type *universal_integer*. The type *universal_integer* is an integer type with a range constraint large enough to encompass every conceivable integer implementation. Since the name *universal_integer* is hidden from the user, he cannot declare variables or otherwise make use of this type.

Additionally, an implementation may provide types LONG_INTEGER and SHORT_INTEGER with larger and smaller ranges than INTEGER. The largest and smallest integer values that are supported by an implementation are SYSTEM'MAX_INT and SYSTEM'MIN_INT respectively.

(2) Universal Float

There is a universal type for floating point numbers. The precision of this type is finite, but sufficiently long to encompass any implemented floating point type. Implementation defined types can be derived from *universal_float* by constraining the precision to the machine implemented precisions. Such types have defined names, for example FLOAT, LONG_FLOAT, etc. The user may define his real types in terms of the machine types, or simply by stating the required precision.

(3) Universal Fixed

There is also a universal type for fixed point numbers. This type is non-discrete like floating point, but the magnitude of errors with this type is *absolute* rather than *relative* (as with floating point types). This absolute magnitude of errors is called the *delta*. The representation of fixed point types depends upon the largest absolute value in the range and the required delta. For instance, a maximum value of 1024 with a delta of 1 will require 10 bits, leaving 6 guard bits on a 16-bit machine (ignoring the sign). The *universal_fixed* type has a finer delta than any implemented fixed type. The effective use of fixed point depends critically upon the proper specification of the range and delta.

The universal types have no defined operation, since the representation cannot be fixed at this level of abstraction. Approximate constants, that is numeric constants with a decimal point or an exponent, are of type *universal_float* or *universal_fixed*, depending upon the context. Hence the machine value used to represent the constant will be within the needed accuracy.

Dividing the literals into integer or real types has obvious advantages in implementation and description. Within an expression, the type of a literal need not be specified by qualification if the context determines the type. However, no implicit conversions are performed between integer numbers and approximate numbers.

5.2 The Integer Types

The operations defined for the integer types are:

Operator	*Meaning*	*Result Type*
+ -	identity and negation	operand type
+ -	addition and subtraction	operand type
*	multiplication	operand type
/	integer division	operand type
mod	remainder on integer division	operand type
**	exponentiation	operand type
relational operators	usual semantics	BOOLEAN

New integer types derived by imposing constraints on INTEGER inherit these operations. The result type is always the type of the operands (except of course for the relational operators). If type LONG_INTEGER is also implemented, then this has the same operations as above, but it is necessary to overload the operations to obtain the required semantics. This step corresponds to the need for the implementation to generate code for such extended integers.

The type SHORT_INTEGER may also be implemented with the above semantics. Note, however, that this type can readily be defined by the user with a type declaration (assuming 16 bits) like:

```
type SHORT_INTEGER is range -32768 .. 32767;
```

The numeric values after **range** are used to determine the appropriate implemented type. For a program requiring large integers we could define

```
type MY_INTEGER is range -100_000 .. 100_000;
```

This type would be implemented with the machine type LONG_INTEGER on a typical 16-bit minicomputer, but with ordinary integers (i.e. INTEGER) on a larger word length machine. If the range cannot be determined at compilation time, then type INTEGER is used.

The operations / and **mod** require explanation. There is no universal agreement on the semantics of these operations for negative operand values. Of course, negative values are not commonly used. Because different machines perform the corresponding operation differently, it is tempting not to define the operation for negative values. This is the approach taken in the axiomatic definition of Pascal. The semantics chosen in the Green language corresponds to division by truncation toward zero (so (-3)/2 = -1). This has the advantage that the usual identity

```
-(A/B) = (-A)/B = A/(-B)
```

also applies to integer division.

The operations / and **mod** are related by

```
A = (A/B)*B + (A mod B)
```

and the sign of the result of the **mod** operation is the same as the sign of A. (Hence A **mod** 10 can be negative). Also the absolute value of the result of the **mod** operation is less than the absolute value of B (which implies that the operation is not defined for B = 0).

The exponentiation operator is only permitted with a positive exponent. Hence, X**(-1) will fail at compilation time as the exponent is not positive. The operation is defined as repeated multiplication of the left hand operand. The number of multiplications is one less than the exponent value (i.e. X**2 = X*X).

Subtypes of integer types can be defined by use of range constraints. Variables of a subtype have the operations of the type but each assignment must conform to the range constraint. One would expect compilers to represent subtypes in the same way as types, but in special cases, the compiler may be able to optimize the representation by utilizing the range constraint.

The absolute value operation is accomplished with the predefined function ABS, which is defined for all numeric types.

5.3 The Real Types

The real types form two classes: floating point types and fixed point types. Both are approximate and are different forms of approximation to the real numbers of mathematics. With floating point types, the error in representing a mathematical value is roughly proportional to its absolute value over a large range. In contrast, the error with a fixed point value has an absolute bound, so that small values have a correspondingly large relative error.

The accuracy constraint specifies bounds on the permitted error in the representation of values: the precision for floating point and the delta for fixed point. The accuracy constraint is handled by the compiler and in consequence there is no exception corresponding to the RANGE_ERROR for a range constraint. Delta is an absolute numeric value and hence should be specified as a fixed point value. Precision is relative and for consistency should be a floating numeric value but there is no doubt that specification by the number of significant decimal digits is more natural. Note that specification of delta in terms of decimal digits is too coarse for a binary machine, and in any case would be unnatural as delta is an absolute value.

The predefined operations provided for floating and fixed types differ in detail in order to reflect correctly the handling of error bounds within a computation. The accuracy constraints determine parameters to a semantic model for the real types which is used to bound errors on the predefined operations.

There are five predefined attributes which apply to both classes of real types. R'FIRST and R'LAST are values of type R which bound all the values of R. The integer value R'BITS gives the number of binary places for the mantissa (floating point) or magnitude (fixed point) in the abstract representation (see 5.3.3 also). R'SMALL and R'LARGE are values of the universal float or universal fixed type which are respectively the smallest positive non-zero value and the largest positive value in the abstract representation.

5.3.1 Floating Point Types

Operations on the floating types are defined at the level of FLOAT and LONG_FLOAT etc. These operations are

Operator	*Meaning*	*Result Type*
+ -	addition and subtraction	operand type
+ -	identity and negation	operand type
*	multiplication	operand type
/	division	operand type
**	exponentiation (right operand any integer type)	type of left operand
relational operators	usual semantics	BOOLEAN

The operators = and /= could have been excluded because their semantics is of doubtful validity, since the representation is approximate. Given a precision of 6 digits, then equality could either mean equality of representation (which would typically be of higher precision) or equality only to 6 digits.

The decision has been to allow equality since it is defined for all other types. The user must be aware that the implemented precision is used and that in consequence code may be non-portable. (The situation is no better with languages other than Green). The semantic model of Brown (see below) handles this.

The types FLOAT and LONG_FLOAT have an implementation defined precision. Derived types can be defined by constraining the range and reducing the precision requirement. The compiler must check (as with other constraints) that these constraints are legitimate. Hence, in practice, at the machine level there will be only one or two implemented precisions. The type declarations may specify the required precision and the predefined machine type name (or other defined floating type).

A user may also define floating point types directly in terms of their precision and range, and this is preferable for portability. In this case the types are mapped on the nearest applicable machine implemented precision. As an example consider the type declarations

```
type MY_SHORT_FLOAT is digits   6  range MIN .. MAX;
type MY_FLOAT       is digits   8  range MIN .. MAX;
type MY_LONG_FLOAT  is digits  10  range MIN .. MAX;
```

On a machine M1 for which the implemented precision provides 7 digits for FLOAT and 14 for LONG_FLOAT these declarations have the same effect as

```
type MY_SHORT_FLOAT is new FLOAT      digits  6  range MIN .. MAX;
type MY_FLOAT       is new LONG_FLOAT digits  8  range MIN .. MAX;
type MY_LONG_FLOAT  is new LONG_FLOAT digits 10  range MIN .. MAX;
```

On a machine M2 for which the implemented precisions provide 8 digits for FLOAT and 16 for long_float, these declarations have the same effect as

```
type MY_SHORT_FLOAT is new FLOAT      digits  6  range MIN .. MAX;
type MY_FLOAT       is new FLOAT      digits  8  range MIN .. MAX;
type MY_LONG_FLOAT  is new LONG_FLOAT digits 10  range MIN .. MAX;
```

If the range constraint is omitted in the type declaration, then the range is inherited from the implemented type. The mathematical library is of course defined in terms of the types FLOAT, LONG_FLOAT, etc., and is hence inherited by the user defined types. If the user writes SQRT(X) where X is of type MY_FLOAT, then on machine M1 the SQRT function defined for LONG_FLOAT will be used whereas on machine M2 it will be the SQRT function defined for FLOAT. Of course the user may always write a special SQRT function, say for type MY_REAL, which may compute a result with exactly 8 digits of precision rather than 14 on machine M1 and 8 on machine M2.

To summarize, the language provides a direct and simple mechanism for achieving efficient use of the available precisions predefined by a given implementation.

The exponentiation operation for floating point operands is defined by repeated multiplication in the same way as with integers. For a negative exponent, the value is the reciprocal of the value with the positive exponent. The exponent can be of any integer type.

The predefined attribute R'DIGITS is the value (of type INTEGER) which appears as the accuracy constraint giving the precision of the type or subtype R.

Example:

Consider the following function, a typical library routine using FLOAT and LONG_FLOAT directly.

```
function DOT_PRODUCT (X,Y: FLOAT_VECTOR) return FLOAT is
   SUM: LONG_FLOAT;
begin
   assert (X'FIRST  = Y'FIRST);
   assert (X'LAST   = Y'LAST);
   for I in X'FIRST .. X'LAST loop
      SUM := SUM + LONG_FLOAT(X(I))*LONG_FLOAT(Y(I));
   end loop;
   return FLOAT(SUM);
end DOT_PRODUCT;
```

If the machine has an instruction which forms the double length product from two single length operands, it is fairly simple for a peephole optimizer to use this instruction in the inner loop (rather than expand each operand and multiply).

If an application requires floating point computation with multiple precisions, then two means can be used to achieve this: the use of subtypes, and the use of types.

Use of Subtypes

To use subtypes, a type must be declared with the largest required precision, for example

```
type MY_REAL is digits 20;
```

Then variables or subtypes can be declared:

```
X : MY_REAL; -- digits 20;
Y : MY_REAL digits 15;
subtype SHORT_REAL is MY_REAL digits 10;
Z1, Z2, Z3 : SHORT_REAL;
```

The operations on MY_REAL are defined for all variables of the type (X, Y, Z1, Z2, Z3). Hence it is not possible to provide an overloaded SQRT function just for SHORT_REAL. Similarly, the error analysis is dependent on the operators for the type MY_REAL.

An optimizing compiler may be able to use single length data representation for each variable, but this depends upon the variables being invisible to other compilation units and on the ability of the compiler to establish that the semantics will be preserved. If an implementation uses call by reference for **in out** parameters, then arrays of SHORT_REAL will have to be handled in the representation of MY_REAL (to avoid the alternative of excessive copying).

Note that the declaration of Y is also an implicit assertion that the precision of MY_REAL is at least 15 digits. This could be useful for defensive programming in large systems. For example, if in a later revision of the program the precision of the type MY_REAL is reduced by more than 5, then an error message will be reported upon recompilation of the declaration of Y.

Use of Types

To use types, each distinct class of numbers would have a different type, with a precision appropriate to the task being performed. Security is better than with the use of subtypes, but all conversions must be explicit. On the other hand, inserting the conversion to another target computer is simple and efficient. This is because each type is mapped separately using only as much precision as necessary. Of course, the efficiency is also high for the initial application computer as well, since even a non-optimizing compiler will map each type onto the appropriate hardware type.

Both cases above assume that the programs have been written well using named types or subtypes. Direct use of FLOAT and LONG_FLOAT is absent, so that no assumption has been made on the precisions of these types.

5.3.2 Fixed Point Types

The definition of the fixed point types is more difficult for several reasons. First, the representation cannot be determined until both the range and delta are known. These two parameters determine the width required in bits and the position of the decimal point. A user could give a representation specification giving only the number of bits needed. Having determined these, the representation is fixed and the operations can be defined. The second problem is that the type resulting from multiplication and division is *universal_fixed*. Since no operations are available on universal types, an expression must be qualified with the required type (or subtype).

In a fixed point type declaration, the value following *delta* and the two range values (which must be provided) are of any fixed point type but must have a value determined at compilation time. In a subtype declaration, the delta value must be larger than that of the type, and the range constraint values must be within the values of the type.

Consider the type

```
type F is delta 0.01 range -100.0 .. 100.0;
```

We assume the target machine to be a 16-bit minicomputer using two's-complement arithmetic. The implemented range would be the next power of two (-128 .. 127) or 7 bits above the decimal point. Similarly 7 bits are required below the decimal point. Hence 15 bits are required (sign, 7 above decimal point, 7 below decimal point), leaving one spare bit at the bottom of the word to provide a (fortuitous) guard bit.

Given two fixed point types F and G then we have the following operations:

Operator	*Meaning*	*Operand Types* *Left*	*Right*	*Result Type*
+ -	identity and negation		F	F
+ -	addition and subtraction	F	F	F
*	integer multiplication	F	any integer type	F
*	integer multiplication	any integer type	F	F
*	fixed multiplication	F	G	*universal fixed*
/	fixed division	F	G	*universal fixed*
/	fixed division by integer	F	any integer type	F
relational operators	usual semantics	F	F	BOOLEAN

The semantics of these operations in terms of the permitted rounding error requires care. The basic source of error is the representation of constants and intermediate results. If EPSILON is half the delta of F (that is, EPSILON = F'DELTA/2), then a constant C is represented by a machine value C1 such that

```
C - EPSILON  <=  C1  <=  C + EPSILON
```

The operations above that yield a result type *universal_fixed* obey a similar inequality:

```
X, Y: F;

X*Y - EPSILON  <=  F(X*Y)  <=  X*Y + EPSILON
X/Y - EPSILON  <=  F(X/Y)  <=  X/Y + EPSILON
```

where the upper and lower limits are calculated mathematically. A value C is representable without error if C1 = C. Computations with such values are exact, except for division and fixed point multiplication. This can easily be seen by thinking in terms of a decimal calculator - no extra digits are needed. Note that integer multiplication is essentially repeated addition, it can overflow but cannot lose accuracy. Note also that integer multiplication by a floating point value is not permitted, since this is not equivalent to repeated addition. Hence the integer operand must be explicitly floated. The user could define this operation if required.

The operations of fixed multiplication and division are essentially in two parts. First, the accurate product is formed (that is, a result of the type *universal_fixed* is obtained). Second, the result must be qualified before being assigned to any variable or being used in further computation. This qualification may imply a loss of accuracy due to the representation in the destination type. The operation of fixed division by an integer operates in an analogous way and is merely provided to avoid excessive explicit type conversions.

The predefined attribute R'DELTA for a fixed point type is a value of type *universal fixed* which is that given in the accuracy constraint of the type or subtype R.

To understand the computational aspects it is simplest to consider a decimal machine. Take a word as being a sign and three digits (SDDD), and consider the following declaration

```
type FRAC is delta 0.001 range -0.999 .. 0.999;
```

This type requires all of the word with the representation S.DDD (i.e. the point next to the far left of the word).

Consider also

```
type LARGE is delta 10.0 range -800.0 .. 800.0;
```

This would ordinarily be implemented as (SDDD.), with one guard digit. Note that in many cases, bit patterns do not give valid values (as with subranges of integers).

Finally, consider

```
type MEDIUM is delta 0.2 range -9.0 .. 9.0;
```

This would have the representation (SD.DD) with more than one guard digit.

We can illustrate the use of these types as follows

```
F1, F2: FRAC;
L1, L2: LARGE;
M1, M2: MEDIUM;

F1 := 0.3333;                    -- last digit 3 lost on conversion to FRAC
                                 -- Now |F1 - 0.3333| <= F1'DELTA

F1 := F1 + 0.1;                  -- 0.1 needs no qualification as the left operand
                                 -- specifies the type (FRAC) of 0.1

F1 := 2*F1;                      -- Now F1 = 0.866

F1 := F1/2;                      -- equivalent to F1 := FRAC(F1/2.0), i.e.
                                 -- integer division avoids qualification

F1 := FRAC(2.3*F1);              -- the constant is represented with as much significance
                                 -- as the other operand. The machine evaluates
                                 -- 2.30*0.433 = 0.99590 (six digit answer) and then
                                 -- rounds the result to 0.996, which is stored in F1.
                                 -- Note that rounding is needed (no guard bit).

L1 := 700.0;                     -- the .0 is necessary. This emphasizes approximation

L1 := LARGE(F1*L1);              -- calculates 700.0*0.996 = 697.20, rounds to 697.0

  ...

L1 := LARGE(F1*L1) + L1;         -- qualification is necessary, and serves
                                 -- to emphasize rounding before addition

L2 := LARGE(F1*L1) + 100.0;      -- qualification is necessary

if L1 > F1 then                  -- not permitted, must be of same type

if L1 > LARGE(L2*F1) then        -- permitted, explicit conversion
```

Fixed point operators = and /= are permitted for the same reason as for floating point.

The user can perform accurate computation with fixed point by ensuring that only exactly representable values are used. In fact, the only source of error is the implied rounding of constants and conversion (which are necessary for multiplication and division). Note that a major source of error in the above may be the representational error of the values rather than in calculating their product.

Example:

A frequent calculation in some numerical applications is the smoothing of an input sequence by means of a running average:

```
OLD_VAL := 0.9*OLD_VAL + 0.1*NEW_VAL;
```

To program this in Green, the right hand side must have the type specified for at least one of the products. An error analysis reveals that a small error in the constant 0.9 will cause a much larger error in OLD_VAL after successive iterations (a constant value of 10.0 as input converges to 9.09 if 0.9 is replaced by 0.89). To avoid this cumulative effect, one can write the following

```
OLD_VAL, NEW_VAL: F;
...
OLD_VAL := OLD_VAL + F(0.1 * (NEW_VAL - OLD_VAL));
```

Example:

Consider the following function for computing the average of an array of components as follows:

```
function AVERAGE (A: FIXED_VECTOR) return F is
   NUM_ITEMS : constant INTEGER := A'LENGTH;
   type SUMF is delta F'DELTA range NUM_ITEMS*F'FIRST .. NUM_ITEMS*F'LAST;
   SUM: SUMF := 0.0;
begin
   for I in A'FIRST .. A'LAST loop
      SUM := SUM + SUMF(A(I));
   end loop;
   return F(SUM/NUM_ITEMS);
end;
```

Here, the type SUMF has a greater range than F to accommodate the larger potential range of values. The explicit conversion inside the loop does not lose accuracy, but the final division will lose potential accuracy. If type F requires nearly a full word, then the type SUMF will be double length. It is very difficult to write an algorithm to obtain the average which avoids double length. Since the size of the array is involved in the type SUMF, this size must be known at compilation time.

5.3.3 A Semantic Model for Approximate Computation

Programming languages do not conventionally define the semantics of floating point arithmetic. However, in this language, with declarations controlling the accuracy of data types, it is highly desirable to do so. Recent work by W. S. Brown [Br 78] makes it possible to describe a model which is both clean in structure and realistic (i.e. it describes the actual behavior of floating point units). In this section, a brief overview is given of the model as needed by the language.

For each type, an abstract representation is defined. Take the declaration:

```
type F is digits 6;
```

This corresponds to 20 binary digits (i.e. F'BITS = 20), and so the abstract representation is of the form of zero or of the form of a sign, twenty binary digits, and an integer exponent. For non-zero values, the most significant binary digit is 1. Such values can be given in a slightly extended form of based numbers, for instance:

```
2     = 2#.1e2
100   = 2#.11001e7
0.25  = 2#.1e-1
```

If the smallest value of the exponent is -99 then

```
F'SMALL = 2#.1e-99
```

If the largest value of the exponent is 99, then

```
F'LARGE = 2#.11111_11111_11111_11111e99.
```

We do not assume that numbers are represented in this fashion, merely that numbers having the numeric values given above are representable in the machine. Brown now develops axioms for the representable numbers and the behavior of a machine number bounded by an interval whose points are representable numbers. These axioms allow the use of higher precision than specified in the declaration, which is essential in Green, since the implemented precision will typically be larger than the declared precision.

For fixed point types, a similar representation is chosen without an exponent. Axioms (not treated by Brown) can now be given which reflect the exact nature of some operations and the approximate nature of others. In addition, because of the obvious correspondence between the abstract representations of all approximate types, conversions can be defined.

These conversions, and some use of subtypes can result in weaker error bounds than those of the type. Consider:

```
type F is digits 6;
X  : F;
Y  : F digits 5;
```

The statement Y := X; allows an implementation to "lose" the three least significant binary digits on the assignment. Hence X := Y; will then mean that the last three bits of X are undefined (i.e. the interval which bounds the value of X is larger than that given by the type).

Example:

Consider an example with a fixed point type since these were not handled by Brown. Take

```
type F is delta 0.01 range -100.0 .. 100.0;
```

Then, as stated above, the representation uses the powers of 2: 64, 32, ... , 1/128 to cover the required dynamic range.

To discuss the semantics, we again write *model* numbers in an obvious extension of based numbers:

```
64 = 2#1000000.0000000
```

Then

```
F'FIRST  = -2#1100100.0000000                          = -100.0
F'LAST   =  2#1100100.0000000                          =  100.0
F'BITS   =  14
F'SMALL  =  1/128 = 2#0000000.0000001                  =  .0078125(<.01)
F'LARGE  =  255 + 127/128 = 2#1111111.1111111  =  255.9921875
-- F'DELTA is not a model number
-- F'FIRST and F'LAST are model numbers in this
-- example but this need not always be the case.
```

Now consider the representation of 2.1, as in the declaration:

```
F1: F := 2.1;
```

The value is bounded by the two model numbers

```
2 + 12/128 = 2#0000010.0001100 = 2.09375
2 + 13/128 = 2#0000010.0001101 = 2.1015625
```

and hence these two values give bounds for F1. On a 20-bit machine, F1 is likely to be represented by the machine value (using the same notation) of

```
2.10009765625 = 8602/4096 = 2#0000010.000110011010 .
```

The error analysis of ordinary computation proceeds similarly. Take:

```
F1 := F1 + 2.0;
```

Here 2.0 is a model number (and hence is represented exactly). So as a result, the bounds for F1 are now 4.09375 .. 4.1015625. If the operands are not model numbers, then the corresponding bounds are used.

The logic with fixed point multiplication and division is slightly different. Take

```
F1 := F(2.1 * F1);
```

Here 2.1 is of *universal_fixed* type, and the context requires that it be represented to the same number of bits as F1.

Hence 2.1 is represented as 2#10.000110011010 or possibly with further bits. But F1 is less accurately represented because of the zero bits at the top of the word. The multiplication provides an accurate product bounded (because of F1) by 2.1*4.09375 .. 2.1*4.1015625. The qualification means that the accuracy of the result is bounded by F'DELTA (or better, F'SMALL). In this case, the bound derived from F1 is large and in consequence the new bound on F1 is about 2.1*F'DELTA. On the other hand, if both of the operands had very tight bounds (or if 2.1 were replaced by anything less than 1 in magnitude) so that the product was bounded by a value less than F'DELTA, then the bound on F1 after the assignment would be F'DELTA.

5.4 Implementation Considerations

Fixed point types can be represented on most machines with one or two machine words. Implementations should not support fixed point types in excess of this length if credible performance is required (and cannot be provided). Such an implementation would be as efficient in time and space as conventional assembler coding (assuming a good register allocation algorithm). As with subranges of integers, tight packing is possible, and could result in a major advantage over floating point.

Good performance depends largely upon the proper specification of the range and delta. For machines with limited arithmetic shifts, a range which excludes negative values would allow the use of logical shifts for scale conversion. All type conversions with fixed types could be accomplished with simple arithmetic shifts and masking. All the operations are likewise straightforward.

With real types there is a problem about the end points of the range with a range constraint. Ordinarily, such values would be in the values permitted. However, a fixed point range 0.0 .. 1.0 on a two's complement machine would not usually want to include 1.0. To avoid giving ranges as 0.0 .. 0.999999999, it seems best not to require that non-zero end values are within the specified range.

A lazy implementation of numeric types that could be used by a diagnostic compiler is as follows. Every value is stored in long floating point format together with a flag indicating if it is integer or real. The long format must be sufficiently long to encompass the longest integer, fixed point and floating point types supported by the implementation. Operations can now be applied to these values, the flag being used to ensure that integer results are correctly rounded to integers (if the floating point hardware does not give integer results from integer values). This implementation method is clearly inefficient, especially for fixed types which are often used as a method of avoiding expensive floating point. However, it illustrates the concept of the abstract value and that the operators have the same meaning for each type.

Although it is theoretically feasible, it is not practical to implement floating point types as fixed point quantities. This is because of the potentially large dynamic range of floating point values, particularly near zero. However, a floating point type whose values were constrained between 1.0 and 2.0 could use fixed point. Such types seem unlikely in practice since even negation is not defined.

With the real types, the language does not specify rounding or truncation, since either choice will be excessively expensive on some machines. However, the user can control its effect by increasing the digits or the delta in the type declaration. Note that a small decrease in the delta could require going from 1 word to 2 words, with consequential performance degradation. With multiplication and division, rounding may be required to preserve the relational inequalities. Exact conversion can only occur between integer types (although many other conversions may not require any rounding). Note that conversions are effectively via universal types so that the specification only involves one type (not all pairs). No conversions are significantly more troublesome than are integer to real, real to integer in (say) Algol 60.

Consider a function F_SQRT for taking the square root of an argument, where the argument and the result are of type

```
type FRACTION is delta D range 0.0 .. 1.0;
```

Given the fixed point quantities

```
X, Y: delta 16.0*D range 0.0 .. 16.0;
```

then one can take the square root by

```
Y := 4.0 * F_SQRT(FRACTION(X/16.0));
```

However, the division by 16 is a shift that corresponds to the converse of the FRACTION type conversion and hence produces no code (assuming reasonable peephole optimization). Of course, the programmer must be aware that this is the case on a binary machine.

The body of F_SQRT (for argument range 0.5 to 1) could be:

```
function F_SQRT(X: FRACTION) return FRACTION is
   APPROX: FRACTION;
begin
   APPROX := 0.5;
   while ABS(APPROX - FRACTION(APPROX/X)) > FRACTION'DELTA loop
      APPROX := FRACTION(0.5 * (APPROX + (FRACTION(APPROX/X))));
   end loop;
   return APPROX;
end F_SQRT;
```

The machine dependence is largely restricted to the declaration of FRACTION, whose range relative to the accuracy would reflect the word length of the machine. Note that if the declared range is 0.0 .. 1.0, then the algorithm may give values equal to 1.0 for arguments near 1.0. This would cause overflow on a two's complement machine. The check for negative arguments is implicit in the type definition.

In evaluating an expression at compilation time, the identification of the operators must be performed. Then expressions involving only literals, other evaluated expressions and predefined operators can be evaluated. The accuracy of real arithmetic may be different from that of the target machine, although both are within that specified in the type declarations.

5.5 Conclusions

The aim of the proposals is to provide the full range of numeric facilities within a secure system of types. This has been achieved by a combination of two techniques. Firstly, use is made of the ability in the language to define new types derived from an existing type from which the new types inherit properties. Secondly, the definition of the accuracy constraints allows a derived type to inherit properties from an existing type of greater accuracy.

The method of inheritance from a named type has been generalized in Green so that the underlying implemented type such as FLOAT or LONG_FLOAT need not be explicitly named. By this means, portable and efficient implementation is ensured. Although several types might be derived from FLOAT (say), they are distinct, and as such increase security since no implicit conversion is permitted. The formulation of the accuracy constraints is vital, since the constraint determines the characteristics of both classes of real types.

Due to recent research, it is possible to define an axiomatic system which gives the minimal properties of approximate computation. These minimal accuracy properties could be exploited by a diagnostic or program analysis system to ensure that the algorithm being used is appropriate. The axiomatic system is realistic in the sense that it can (and must) be applied to existing floating point implementations.

The numeric types available to the programmer are derived from those defined in the implementation. This guarantees that the effiency of the resulting code is directly related to that of the implementation, which, in the case of floating point, could be hardware, firmware or software. Fixed point types are not predefined in the standard environment, but acquire their properties on declaration. However, in terms of code generation these properties involve little more than that of the integers and hence the performance should be high.

Portability cannot be guaranteed by the language because it is not possible for a program to be completely isolated from dependencies on the underlying hardware. However, such dependencies are limited to the attributes of the predefined numeric types, and properties of the implemented real types which cannot be derived from the axiomatic system (such as the radix of the floating point system).

To conclude, the numeric types in Green provide facilities clearly needed by the envisaged applications. A rigorous axiomatic system is proposed to handle approximate computation. Most importantly, portability and efficiency are not sacrificed.

6. Access Types

6.1 Introduction

The notion of access type encompasses the concept of objects that are dynamically created during the execution of a program. In general, neither the number of such objects in existence at any given time, nor the names of those objects, can be fixed in advance.

The inclusion of such a feature in a language raises what are traditionally some of the most difficult issues in language design, and indeed in programming. Accordingly, the first section of this chapter is devoted to an overview of the issues involved in the area. This will serve as background for an exposition of the approach adopted in the Green language. In many places in this chapter we have borrowed concepts and even wordings inspired by the Euclid report.

6.2 Overview of the Issues

The main problems usually encountered with access types fall into two categories:

- Conceptual aspects
- Reliability, efficiency, and implementation issues.

We first discuss these problems and then define the desirable goals for a formulation of access types.

6.2.1 Conceptual Aspects

Variables of a program can be classified into two categories: *static* variables and *dynamic* variables.

Static variables are declared in a program and are containers for values. Each static variable has a name that is used to denote either the container or the value, depending on the context where the name appears. The name of a static variable is introduced by its declaration, together with its type. The variable exists during the entire lifetime of the program unit to which it is local. Such variables are said to be static since their lifetime is determined by the static structure of the program.

In contrast, dynamic variables are created dynamically without any relation to the static program structure, by the execution of so called *allocators*. In general, the number of dynamically created variables may be unpredictable and it will be impossible to introduce all the names of dynamic variables by declarations. Hence the *internal* name or reference that is produced by the dynamic allocation cannot be used explicitly to designate the newly allocated variable.

To treat this problem, one usually defines by declaration a number of indirect names that can be used to access the internal names of dynamically allocated variables. These indirect names are variables that may be used to access the internal names of different dynamic variables at successive stages of execution. Hence, indirect names will be called *access* variables throughout the remainder of this chapter. Access variables can also appear as components of dynamic variables.

Four important consequences follow from the fact that access variables contain internal names:

(1) Access variables may be used to describe relations that change over time.

(2) The same internal name may be contained in several access variables, with the consequence that they enable access to the same dynamic variable.

(3) Since access variables may contain different internal names at successive stages of the program execution, a given dynamic variable may become inaccessible. A dynamic variable will remain accessible if its internal name is contained in a statically allocated access variable or in an access variable that is a component of another accessible dynamic variable.

(4) Since an access variable does not contain any internal name until its first allocation or assignment, there must be a special null value corresponding to no internal name (**none** in Simula, **nil** in Algol 68 and Lisp, **null** in this language). This value is also required for describing partial relations.

Sharing and the possibility of inaccessibility are thus two of the classical difficulties of access types.

A third classical difficulty is the well-known problem of dereferencing. Consider the name of an access variable; this name may stand for (or provide access to) several different things.

- The name of the (static) access variable
- The content of the access variable (that is, its value: an internal name)
- The content of the dynamic variable designated by this internal name.

The first two possibilities (name or content) also exist for static variables. Most languages (Bliss being the exception) have the same notation in the two cases, and make a distinction by context. The third possibility, however, only exists for access variables, and the solutions offered by programming languages are very diverse.

Two issues arise:

(1) For assignments, it must be clear whether the assignment refers to the access variables (access assignment), or to the dynamic variables they designate (value assignment). This distinction is essential and has been treated differently in most languages.

(2) For component selection, i.e., for denoting a component of a dynamically created record, there is no possible ambiguity. Nevertheless some languages have chosen to make dereferencing explicit even in this case.

The diversity of the solutions adopted by several languages is a clear indication of the conceptual difficulties involved. We illustrate this diversity with the following example, where X and Y are access variables and AGE is a component of the dynamic record variable (For the Algol 68 formulation, T is assumed to be the mode of the record values; the Simula example extends the possibilities offered for texts).

language	*access assignment*	*value assignment*	*component selection*
Simula	X :- Y;	X := Y;	X.AGE
Algol 68	X := Y;	T(X) := Y;	X.AGE
Pascal	X := Y;	X↑ := Y↑	X↑.AGE
Green	X := Y;	X.**all** := Y.**all**;	X.AGE

A final conceptual difficulty in defining access types is the notion of constant access objects. Suppose the name of an access object is declared to be constant. Several alternative interpretations could be given to such a declaration.

(a) The access value (an internal name) is constant. This means that it always designates the same dynamic object. The value of the latter, however, could vary.

(b) The access value is itself variable, but it may only be used to read the components of a designated object.

(c) The access value is constant and it may only be used to read the components of the designated object. Note however that we cannot infer that the dynamic object designated by such a constant is itself constant if other variables designate the same dynamic object.

Some languages, including Mary [R 74] and Lis, have provided alternate syntaxes for all three forms of semantics. The third meaning however is the most useful and hence, for the sake of simplicity, should be retained as the unique meaning of constancy. Note that with this semantics, an access constant only has a read permit for the components of the designated object. Hence an access constant cannot be assigned to an access variable since a variable would have both a read and a write permit and only the read permit can be obtained from the constant.

6.2.2 Reliability, Efficiency, and Implementation Issues

When a dynamic variable becomes inaccessible, the corresponding space may, at least theoretically, be recovered for other uses without any risk. This operation, classically called garbage collection, has been used in languages such as Lisp, Simula, and Algol 68. It is however rather costly and cannot be used effectively for real time systems, since it may occur unpredictably at critical times.

For this reason several languages in the system programming area (including Lis and Euclid) try to achieve a better control on the storage management for dynamic variables. This means that such languages offer the opportunity to define the workings of object allocation within the language itself. Similarly they permit an explicit deallocation statement which can also be defined within the language itself.

Clearly such operations cannot be written without violating strong typing. In addition, the availability of explicit deallocation opens the possibility of dangling access values, since a programmer may deallocate a dynamic variable whose internal name is still contained by other access variables.

Confronted with this dilemma between reliability and efficiency, a possible answer is to choose reliability and to accept the possibility that access types might not be used in programs that are time critical. However, there are cases where access types should be used, precisely because the application considered is time critical. We illustrate this point.

Assume that we need to search a circular list for an item with a particular content. A formulation using an array might look as follows.

```
subtype INDEX is INTEGER range 1 .. 1000;

type ITEM is
   record
      SUCC, PRED : INDEX;
      CONTENT    : INTEGER;
   end record;

TABLE : array (INDEX'FIRST .. INDEX'LAST) of ITEM;
HEAD, NEXT : INDEX;
SUM : INTEGER;
```

The algorithm for adding the contents of the successors of HEAD may be written as a while loop:

```
SUM  := 0;
NEXT := TABLE(HEAD).SUCC;
while NEXT /= HEAD loop
   SUM  := SUM + TABLE(NEXT).CONTENT;
   NEXT := TABLE(NEXT).SUCC;
end loop;
```

Clearly, the above formulation attempts to use index values in order to express relations and does not achieve this with the elegance and readability offered by access variables. The main point, however, is that the index computation involved in accessing the array element TABLE(NEXT) at each iteration may be a drawback, especially on minicomputers where multiplication is rather slow.

The alternate formulation with access variables (declarations omitted) is given below:

```
SUM  := 0;
NEXT := HEAD.SUCC;
while NEXT /= HEAD loop
   SUM  := SUM + NEXT.CONTENT;
   NEXT := NEXT.SUCC;
end loop;
```

This solution is more readable (it does not require mention of names such as TABLE that are irrelevant to the logic of the algorithm), and also more efficient since no index calculation is involved.

In general, when access variables are used, address computations will be done once, at the time of dynamic allocation. Thereafter access variables can only be assigned to other access variables or used to access the dynamic variables. This however does not involve address computations. On the contrary, when indices are used, address computations must be redone for every access.

6.2.3 Goals For a Formulation of Access Types

As shown by the previous example, one of the motivations for access variables is efficiency. As a consequence we must be able to use them in time critical applications. In this case, however, we must provide a form of access variables that does not result in garbage collection with the associated costs and unpredictability. Naturally this does not exclude the possibility of more elaborate storage management strategies in applications that are not time critical.

The needs of efficiency being thus satisfied, it remains that reliability should be a major goal in the formulation of access types, especially in view of the conceptual difficulties they raise. A safe formulation of access types should hence have several important properties:

- There must be a **null** value for access variables. The null value cannot be dereferenced and any attempt to do so should result in the exception ACCESS_ERROR (on many computers this is achievable without any runtime cost by selecting a protected address as the internal value of **null**).

- Access variables should be typed (as in Pascal) so that access variables can only designate dynamic variables of a single type.

- There should be a language defined operation (the allocator) that creates a dynamic object and delivers its internal name, the access value. On the other hand, there should be no operation for explicit deallocation of a dynamic variable (to avoid dangling access values).

- There should be a clear distinction between access types and other types. In particular, there should be no possibility for an access variable to denote a static variable (again, to avoid dangling access values).

6.3 Presentation of Access Types

The presentation of the properties of access types in the Green language will cover the following topics:

- How to declare access types
- The collection of dynamic variables implied by the declaration of an access type
- How to declare access variables and constants
- How to allocate a dynamic record variable
- Component and value assignments
- Recursive access types
- Procedures and functions with parameters belonging to an access type
- The control of storage management for a collection of dynamic variables

6.3.1 Declaration of Access Types and Subtypes

Access variables like other variables in the Green language are typed, and belong to a so-called access type. The example below shows a declaration of an (ordinary) record type followed by the declaration of an access type:

```
type PERSON_VALUE is
  record
    AGE  : INTEGER range 0 .. 130 ;
    SEX  : (MALE, FEMALE);
  end record;

type PERSON is access PERSON_VALUE;
```

In this example, PERSON_VALUE is declared as a (static) record type. Hence static variables of this type can be declared as usual. The access type PERSON is declared in terms of the access type definition **access** PERSON_VALUE. This means that access variables of type PERSON can only refer to dynamically allocated record variables of type PERSON_VALUE.

It is of course possible to copy the value of a dynamically allocated PERSON_VALUE into a static variable of this type and vice versa. Note, however that there is no way for an access variable of type PERSON to designate a static variable of type PERSON_VALUE.

In practice it is not necessary to name the type of the dynamic variables. Thus the previous declaration can be achieved in a single step

```
type PERSON is access
  record
    AGE  : INTEGER range 0 .. 130 ;
    SEX  : (MALE, FEMALE);
  end record;
```

The type of the dynamic variables can be any type other than an access type itself. For example it can be an array type, as in

```
type TEXT is access STRING;
```

It is possible to declare a subtype of an access type, where constraints can be imposed on the dynamically allocated objects in the subtype declaration. Thus the subtype LINE defined below corresponds to dynamically allocated strings of 80 characters:

```
subtype LINE is TEXT (1 .. 80);
```

Note that although it is possible to imagine an access type definition in a variable declaration, such a declaration would be rather useless. In practice, access type definitions will always appear in type declarations. Hence access types are always named.

6.3.2 Collections of Dynamic Variables

Conceptually it is important to realize that each access type declaration implicitly defines a collection of potential dynamic variables. The actual collection will be built during program execution as allocators are executed. Its lifetime cannot be longer than that of the program unit in which the access type definition is provided.

Collections in the Green language are implicit and cannot be named (unlike those in early Pascal, Lis, and Euclid). The collections associated with different access types are always disjoint, i.e., two access variables of different access types are guaranteed to contain the internal names of dynamic records in different collections.

Finally, the collection associated with a given access type must be considered as part of the global environment that is accessible in the scope of the access type declaration.

6.3.3 Access Variables, Allocators, and Access Constants

Access variables are declared in the usual way and may be initialized in their declaration, for instance with the value of other previously declared access variables or with the special value **null** representing no internal name. For example, consider

```
YOU, HIM, HER : PERSON;
SOMEONE : PERSON := null;
```

An allocator creates a dynamic variable and assigns its internal name to an access variable:

```
YOU := new PERSON(AGE => 30, SEX => FEMALE);
```

The allocator mentions the access type name (or access subtype name) and contains an aggregate defining the initial value of the components of the dynamically allocated variable.

The constraints applicable to a dynamically allocated object are established when the allocator is evaluated and cannot be modified during the lifetime of the dynamic object. In the case of a dynamic array, this means that the bounds of such an array cannot be modified. Consider

```
MESSAGE : TEXT := new TEXT(1 .. 80 => " ");
```

It is possible to modify the character values of the string designated by MESSAGE, but the bounds of this string remain those that are fixed at allocation time, i.e. 1 and 80.

Similarly, for a record type with variants, the discriminant values established at allocation time cannot be modified:

```
type BUFFER is access
  record
    LENGTH  : constant INTEGER range 1 .. 1000;
    POS     : INTEGER range 0 .. 1000;
    ALPHA   : array (1 .. LENGTH) of CHARACTER;
  end record;

B : BUFFER;
B := new BUFFER (LENGTH => 40, POS => 0, ALPHA => (1..40 => "*"));
```

The component LENGTH is a discriminant used to establish the upper bound of the array ALPHA. Hence the value of LENGTH, once initialized by the allocator, cannot be changed thereafter (not even by a global record assignment to the dynamic record variable). As a consequence, only the size actually required by the dynamic object need be allocated.

Declarations of access constants are given in the usual way. The access value (an internal name) contained by an access constant cannot be changed, and the constant name can only be used to read the components of the designated dynamic object. Consider, for example, the constant declarations:

```
YOU_NOW  : constant PERSON := YOU;

DAY_NAME : constant array (1 .. 7) of TEXT :=
  (new TEXT("MONDAY"),    new TEXT("TUESDAY"),   new TEXT("WEDNESDAY"),
   new TEXT("THURSDAY"),  new TEXT("FRIDAY"),    new TEXT("SATURDAY"),
   new TEXT("SUNDAY"));
```

The constant YOU_NOW contains the internal name of the dynamic record designated by YOU at the time of the initialization. It means that YOU_NOW will always contain this access value even if YOU is updated at a later time. Using the constant YOU_NOW one can read but not modify the components of the corresponding person. It does not necessarily mean however that these components remain constant if variables such as YOU are used to modify the components.

The array DAY_NAME is a constant array, hence its components are constant access values. In this case the internal names associated with each access constant are obtained from allocators. Since an access constant cannot be assigned to an access variable, we can infer that the characters of the strings denoted by DAY_NAME cannot be modified. Hence it would even be legitimate for a translator to perform the corresponding allocations statically (at translation time).

6.3.4 Component Selection, Indexed Components and Value Assignments

In the previous example, the content of YOU is the internal name of a dynamically allocated record variable. The usual syntax of component selection is used as if YOU were the record variable itself (i.e. dereferencing is implicit for component selection):

```
YOU.AGE        -- a variable of type INTEGER

YOU.SEX        -- a variable of type (MALE, FEMALE)
```

Similarly, we can use the normal selection syntax to designate the entire (dynamic) record object. Thus YOU.**all** is an object of type PERSON_VALUE such that the following condition is true:

```
YOU.all = (AGE => YOU.AGE, SEX => YOU.SEX);
```

This notation can also appear in an allocator, as in the assignment statement

```
HER := new PERSON(YOU.all);
```

Finally the same notation may be used for value assignments. Remember that if YOU and HER contain internal names of dynamically allocated record variables, then after the assignment

```
YOU := HER;
```

the two access variables contain the same internal name. In contrast, the value assignment for copying the value of the dynamic record designated by HER into the dynamic record designated by YOU is written

```
YOU.all := HER.all;
```

Such value assignments are always possible between dynamic record variables without variants. With variants, they are legal only if the discriminants of the variables are identical. This must be checked (in many cases at execution time).

Indexed components for arrays denoted by access types are written exactly as in the case of static arrays (this means that dereferencing is also implicit for indexing). Thus we can write

```
MESSAGE(1) := "*";
MESSAGE(11 .. 16) := DAY_NAME(1)(1..6);
MESSAGE(21 .. 27) := "MORNING";
```

Note finally that the notation X.**all**, denoting the dynamic object designated by X, can be used for all dynamic objects, whether they are records, arrays, or scalars.

6.3.5 Recursive Access Types

As stated earlier, the components of a dynamic record can be of any type, including dynamic types. This introduces the possibility of *recursive* access types with components designating dynamic records of the same collection. As an example, the access type PERSON may be extended as follows:

```
type PERSON is access
  record
    AGE      : INTEGER range 0 .. 130;
    SEX      : (MALE,  FEMALE);
    SPOUSE   : PERSON;
  end record;
```

The variable HIM.SPOUSE can be assigned another PERSON variable;

```
HIM.SPOUSE := HER;
```

Furthermore, HIM.SPOUSE.AGE is an INTEGER variable, HIM.SPOUSE.SPOUSE is a variable of type PERSON, and so forth.

This kind of *recursion* in access type declarations may involve more than one access type. In such cases it is first necessary to provide an incomplete declaration of any access type whose name is mentioned before the occurrence of its full declaration. This is shown by the following pair of access types:

```
type CAR;  -- incomplete predeclaration of CAR

type PERSON is access
  record
    NAME     : STRING;
    AGE      : INTEGER range 0 .. 130;
    SEX      : (MALE, FEMALE);
    SPOUSE   : PERSON;
    VEHICLE  : CAR;
  end record;

type CAR is access
  record
    NUMBER   : INTEGER;
    OWNER    : PERSON;
  end record;
```

6.3.6 Access Objects as Parameters

Like other variables, access variables can be used as parameters, and parameter modes have their usual meaning. Moreover, one must bear in mind that an access constant can only be used to read the components of the object designated by the contained internal name. This rule also applies to **in** parameters. Hence if components of an access parameter must be updated, it must be declared with the mode **in out**.

For functions, the parameters must as usual be **in** parameters. In addition the collection of objects so far allocated must also be considered as an implicit **in** parameter of the function. Consequently, it is not possible to modify a component of a dynamic record within the function. Similarly, it is not possible to evaluate an allocator for this access type within the function.

This does not prevent assignments of local access variables and, more generally, operations on access types local to the function. As an example consider the declarations for the lists of section 6.2.2.

```
type ITEM is access
   record
      SUCC, PRED : ITEM;
      CONTENT    : INTEGER;
   end record;
```

A function CARDINAL counting the elements in a given circular list can be written as follows:

```
function CARDINAL(HEAD: ITEM) return INTEGER is
   -- The head is not counted as a list element
   -- For an empty list HEAD.SUCC = HEAD.PRED = HEAD

   NEXT  : ITEM    := HEAD.SUCC;
   COUNT : INTEGER := 0;

begin
   while NEXT /= HEAD loop
      NEXT  := NEXT.SUCC;
      COUNT := COUNT + 1;
   end loop;
   return COUNT;
end;
```

Note that assignment to the local access variable NEXT does not modify the collection of elements. Note also that the components of the dynamic records may be read, such as in NEXT.SUCC.

When allocators have to be executed, value returning procedures (or just procedures) must be used.

6.3.7 Storage Management for Access Types

Unless specified otherwise, the collection of dynamic objects associated with an access type will be allocated in a global heap (and may be garbage collected in some implementations). For time critical applications, however, it is possible to specify an upper bound for the space needed for a given collection. The corresponding space can then be reserved globally when the access type definition is elaborated. Subsequently, when leaving the program unit enclosing the access type definition, the space corresponding to the collection may be recovered since the contained objects are no longer accessible. Such an upper bound is indicated by a length specification:

```
MAX_NUM : constant INTEGER := 1000;
...
for   PERSON use MAX_NUM*PERSON'SIZE;
```

The expression provided after the reserved word **use** is the size in bits of the storage area to be reserved for the collection of persons. Naturally if we have an estimate of the maximum number of persons (MAX_NUM) to be allocated, the expression should be formulated with the attribute PERSON'SIZE.

A collection for which such a length specification has been given behaves as a (static) array in so far as storage allocation is concerned. The objects are allocated within this static storage area by allocators, and remain allocated until the collection disappears when leaving the program unit where the access type definition appears. The exception STORAGE_OVERFLOW is raised when the space reserved is exhausted.

Such collections may be allocated either on the stack or on the heap. They have several advantages. In terms of storage management they have a cost comparable to that of arrays. In addition they offer both the notational advantages and the addressing efficiency of access variables.

7. Subprograms

Subprograms can be functions or procedures. The form of these program units is quite traditional, following from Algol 60. Nevertheless the design of a subprogram facility raises difficult issues in terms of the organization of the program text, the definition of the parameter mechanism, the nature of functions, and overloading. These issues are discussed in separate subsections.

7.1 Subprogram Declarations and Subprogram Bodies

The textual representation of subprogram bodies is largely classical as shown in the example below:

```
procedure PUSH(E : in ELEMENT_TYPE; S : in out STACK) is
begin
   if S.INDEX = S.SIZE then
      raise STACK_OVERFLOW;
   else
      S.INDEX := S.INDEX + 1;
      S.SPACE(S.INDEX) := E;
   end if;
end PUSH;
```

However, the Green language allows the subprogram declaration to be separated from the subprogram body. As an example, the subprogram declaration

```
procedure PUSH(E : in ELEMENT_TYPE; S : in out STACK);
```

may appear grouped with other subprogram, variable, constant, and type declarations in a given declarative part, whereas its body may appear later in the list of bodies of the declarative part.

The main reason for permitting such separation is readability. If the body and the specification appear together (as in Algol 60), the potentially large body is mixed with the smaller interface specification. The specification may be hard for the reader to find, especially when examining a program with a large number of subprograms spread over several pages of text. In addition, an isolated variable declaration between two large subprograms is a well-known source of error in Algol 60 (the neglected variable may hide an outer variable that is in consequence never updated). These inconveniences are avoided in the Green language. In a declarative part the following elements, if present, appear in this order:

- use clause
- list of declarations
- list of representation specifications
- list of bodies

The user has thus the option to regroup all subprogram declarations within a small space of text, thereby providing an immediate overview of all subprograms that are local to a given scope. In addition, the above syntax forbids the declaration of a variable between two subprogram bodies, since the bodies appear last.

The split of the subprogram declaration from its body is a convenience for large subprograms, but it is a necessity for subprograms declared in the visible part of a package, and for subprograms that are mutually recursive. Requiring a split in all cases, including small subprograms, would however add verbosity without compensating advantages. In the Green language the decision to split is therefore left to the programmer, except in the cases just mentioned where it is necessary.

Although this important textual issue is left to the programmer, no semantic problems are involved since the information provided by the subprogram declaration is repeated in full in the subprogram body.

7.2 Parameter Modes

Three parameter modes, **in**, **out**, and **in out** are provided in the Green language. These modes are defined in terms of their abstract behavior (i.e. without referring to their implementation) as follows:

in — Within the subprogram, the parameter acts as a local constant whose value is provided by the corresponding actual parameter.

out — Within the subprogram, the parameter acts as a local variable; its value is assigned to the corresponding actual parameter as a result of the execution of the subprogram.

in out — Within the subprogram, the parameter acts as a local variable, and permits access and assignment to the corresponding actual parameter.

For each parameter mode, the implementation may choose to provide access to the corresponding actual parameter either (a) throughout the call (i.e. by *reference*) or (b) by copying it before or after the call, or both, as appropriate: before the call for **in** parameters, upon return for **out** parameters, both for **in out** parameters.

This translator choice may be influenced by the size of the objects considered. For large objects an implementation by reference is often more efficient. On the other hand, for objects that are smaller than the machine's addressable storage units, copying will usually be more efficient.

The problem of access to small objects is indeed severe and may be illustrated by the problem of the access to parameters which are boolean components of records. Although such components have the same type (BOOLEAN) there is no guarantee that they will always be found in the same bit position within a record.

The two known alternatives to copying in the case of **in out** parameters are equally unacceptable:

- One might associate an implicit subprogram (a *thunk*) with each actual boolean parameter. This is both complex and inefficient.
- One might, as in Pascal, forbid components of packed records as actual parameters for **in out** formal parameters, and adopt otherwise a standard representation for all small objects, for example each boolean component on an addressable storage unit. The problem with this solution is that for all practical purposes it would force programmers to use representation specifications in too many cases; the translator's choice being too often unacceptable except on machines with small addressable units. In addition, the program validity would depend on whether or not a representation specification were given.

In *normal* situations the semantics of a program will not be affected by the fact that parameter passing is implemented by reference or by copying. The *abnormal* situations are:

(a) Shared variables:

Consider the subprogram

```
procedure P(X : in T);
```

and assume that the implementation has chosen to implement parameter passing by reference. Then a call P(S) where S is a shared variable must be compiled with some precaution. The code of the procedure P relies on the fact that X is constant and this might not be the case for a shared variable such as S. The solution in this case is to create a local copy of S on the calling side:

```
local_copy := S;
P(local_copy);
```

(b) Exceptions

If a subprogram execution is abnormally terminated, by an exception, then **out** and **in out** parameters may have been updated if implemented by reference, and will be unchanged if implemented by copying.

It would certainly be possible to complicate the runtime exception executor so that copying back of current values is achieved in case of termination by an exception, but it is not worth the price. Consider for example:

```
procedure P(X : out INTEGER) is
begin
   -- (1)
   X := ...;
   -- (2)
end;
  ...
P(U);
```

If P is abnormally terminated by an exception, the only information that the caller has is the nature of the exception. He does not usually know whether it occurred during (1) or (2) or even during the result assignment. So the uncertainty introduced by not knowing the implementation is of the same order as the uncertainty that already exists about the exact point of the exception. In addition, when a user writes P(U) where U is an **out** parameter, he expects the value of U to be changed. So it does not matter much if this value is changed during the call or only at the end. If the user wants to reuse the previous value of U in the case that P is terminated by an exception, the only logical way to do so is to assign its value to another variable before the call.

(c) Aliasing

If illegal aliasing is used then the results may differ for reference implementations and copy implementations. For example consider

```
procedure P(X : in out INTEGER) is
begin
   X := X + 1;
   X := X + A;
end;
```

Since A is accessed within the body of P, a call such as P(A) would be illegal since there would be two possible access paths to A (directly or through the parameter X). Assuming A = 2 before a call P(A), the value of A upon return would be 5 for an implementation by copying and 6 for an implementation by reference.

These are the only possible cases where reference and copy implementations might differ. The problem with shared variables can be solved by creating local copies. For exceptions nothing need be done since the uncertainty introduced is of the same order as the usual uncertainty about the localization of the exception. Some cases of aliasing will be detected by the compilers but some more complex cases must remain undetected. In any case aliasing is a programming error. Any program containing such an error may deliver different results (but erroneous ones in any case) on different machines.

During this design we considered (and rejected) several alternative views to this abstract formulation of the parameter passing modes. For example an implementation oriented formulation of modes could be defined in terms of the mechanisms involved, i.e. copying or reference. However, if the same capabilities are to be offered it leads to more modes (constant by copying, constant by reference, variable by copying before and after, variable by reference, result by copying, result by reference). Although only a subset of them might be provided, it is critical for reliability and efficiency to be able to pass an array by reference and nevertheless deny the right to modify its elements. Apart from its complexity, such a formulation would force the programmer to think in terms of (and be aware of) the representation of objects, and would therefore compromise portability.

We consider the formulation of the parameter passing modes **in**, **out**, and **in out** in terms of their abstract behavior to be much simpler and preferable. Programs that rely on some assumption concerning the implementation of parameter passing modes are erroneous.

7.3 Parameter Passing Conventions

Two parameter conventions need to be considered. The usual positional notation is almost universal. However, with more than three or four parameters it is hard to follow the text, and not even the parameter mode is apparent. Following several authors [Fr 77, Har 76, IRHC 74] and common usage in many control languages, the Green language also permits an alternative form of parameter passing in which the associations are specified on a name basis. Placing the formal parameter on the left and the corresponding actual parameter on the right of a parameter association it is possible to provide more readable procedure calls. These provide knowledge of the parameter modes, and hence of the possible side effects on an actual parameter, such as on MY_-FILE below:

```
CREATE(FILE :=: MY_FILE, NAME := "finaltext.Feb.15");
```

Where long parameter lists are common and have default values, such as in the job control area, this form of named parameter associations provides especially high readability. It may be used in conjunction with the default value facility available for an **in** parameter if no explicit value is provided within the call.

As an example, a simulation package may declare the procedure ACTIVATE as follows:

```
procedure ACTIVATE( PROCESS  : in PROCESS_NAME;
                    AFTER    : in PROCESS_NAME := NO_PROCESS;
                    WAIT     : in TIME := 0.0;
                    PRIOR    : in BOOLEAN := FALSE);
```

As shown in this declaration, the parameter PROCESS must be provided in all calls (because no default value is given). On the other hand the parameters AFTER, WAIT and PRIOR may be omitted. Thus the two following calls of ACTIVATE are equivalent:

```
ACTIVATE(PROCESS := X, AFTER := NO_PROCESS, WAIT := 0.0, PRIOR := FALSE);
ACTIVATE(PROCESS := X);
```

Clearly in many contexts the order of parameters is either highly conventional (such as for coordinate systems) or immaterial (such as in MAX(X,Y)). Hence the Green language admits both conventions. The classical positional convention may be used whenever the programmer feels that named parameters would add verbosity without any gain in readability.

The two conventions may also be used concurrently, with positional parameters appearing first, that is, once naming is used the rest of the call must use naming. This allows the default value mechanism to be used even when a positional notation is desirable, as in the following graph plotting and simulation examples:

```
MOVE_PEN(X1, Y1, LINE  := THICK);
MOVE_PEN(X2, Y2, PEN   := UP);

ACTIVATE(X);
ACTIVATE(X, AFTER  := Y);
ACTIVATE(X, WAIT   := 50.0*SECONDS, PRIOR := TRUE);
```

As shown in this last example, the naming conventions may be used in conjunction with the default parameters to provide a high degree of expressivity and readability that could only be achieved at the expense of a predefined syntax for the activate primitive in Simula [DNM 69].

7.4 Function Subprograms

The purpose of a function is to calculate a value. This is the conventional mathematical meaning of a function. Small functions to access complex data structures are an essential feature of modern software. Such functions can not only hide irrelevant parts of the data structure but can provide a cleaner interface to the outside world.

Although the mathematical origin of the function concept is clear, its incorporation into a programming language can lead to several different solutions depending on the operations that are allowed on variables. Different levels of restrictions can be considered leading to different concepts of functions:

(1) Reading global variables is not allowed.

(2) Reading global variables is allowed but updating them is not.

(3) Updating global variables is allowed provided that the subprogram is not called at points where these variables are visible.

(4) Updating global variables is allowed without restrictions.

The first level corresponds to the mathematical notion of function; there are no implicit parameters in the form of global variables. Consequently two function calls with the same arguments always deliver the same result. However, the class of cases in which such functions can be used is rather limited and does not justify its identification as a feature of a programming language.

The second level is more common, and it also has interesting mathematical properties that can be used for code optimization. For example, if F and G are two such functions delivering results of a given type (assuming * to be commutative):

F + F is equal to 2 * F
F * G is equal to G * F

Functions of this form are provided in the Green language. They are indicated by the reserved word **function** in their declaration. For example:

```
function CARDINAL(HEAD : ELEMENT) return INTEGER;
```

Such functions must not have side effects. This requires:

(a) no update of global variables.

(b) only parameters with **in** mode.

(c) no call to procedures which update globals, directly or indirectly.

(d) no updating of components of access variables.

(e) no allocation statement within a function body.

These checks can be performed by the translator. For procedures it requires checking that the called procedures have the same no-side-effect property in a transitive manner. The main difficulty is for separately compiled procedures.

This form of functions may not perform input-output since this is a side-effect. One may object that this prevents instrumentation and debugging of such functions. A pragmatic attitude must be taken here. It is to be expected that the environment of tools developed around this language will provide facilities for instrumentation and for tracing values. Such facilities can be specified as pragmas recognized by the translator and hence their actions will not be considered as side effects.

The third level is illustrated by subprograms such as random number generators or *memo functions*, which modify their environment. This is provided in the Green language by *value returning* procedures, i.e. by procedures whose specifications include a result:

```
procedure DRAW(CHANCES : REAL range 0.0 .. 1.0) return BOOLEAN;
```

Such procedures may be called within expressions and obviously do not have the aforementioned properties of functions. For example:

```
DRAW(0.5) or DRAW(0.5)
```

is not necessarily equal to DRAW(0.5). Calls to value returning procedures within an expression are evaluated in the order in which they appear. Such calls are not allowed in expressions defining the constituents of types or of constraints.

The fourth level, with arbitrary side-effects, would undermine the advantages of the functional approach to software. In addition it would complicate the semantics of all language constructs where expressions involving such calls may occur. Hence this form of functions is not provided.

7.5 Overloading

At any point in a program, several subprograms declared with the same identifier or operator symbol may be visible, without hiding each other as would be the case for variables with the same identifier. Such subprograms (and their identifier or operator symbol) are said to be *overloaded*. Overloading of subprograms with the same designator (the identifier or operator) is possible if the subprogram specifications are different in other respects. For identical specifications the usual rules of redeclaration apply: redeclaration in a nested unit hides the outer declaration; redeclaration is not allowed within the same declarative part.

Overloading is also possible for literals. An enumeration literal may denote values of different enumeration types; a number may belong to several numeric types. For example, 1.5 may denote a value of a fixed point type or a value of a floating point type.

There are several reasons for permitting these forms of overloading. Above all, the careful use of overloading can be a valuable asset for readability since it offers an additional degree of freedom for the choice of suggestive identifiers that convey the properties of the entities they denote.

Most programming languages already use the same symbol "+" to denote addition for integer values and for floating point values (although addition is not commutative for floating point types because of the approximate nature of computations). Since addition has similar properties for rational and for complex numbers, it is only natural to reuse the same operator when defining these two types. Similarly, it is natural to use the same name for an output operation such as PUT, whether it is used for an argument of type integer, real or character, or for a user defined type.

Extension or overloading of the meaning of an operator is in general quite valuable for user defined types. Such types will normally be defined in packages together with their associated operations. The writers of these packages can certainly not foresee all possible uses. In particular they may not (and should not) be aware of the fact that these packages are used in contexts where some operations appear as overloaded. The writers should be free in their choice of suggestive names but the users should not be burdened by inconvenient choices over which they have no influence. Overloading achieves this separation. Similar arguments apply for overloading of enumeration literals.

Overloading of subprograms and overloading of enumeration literals are therefore allowed in the Green language. On the other hand, overloading of names is not provided for other entities such as variables, constants, types, etc. As we shall see, the identification of occurrences of overloaded subprograms uses the contextual information provided by the types of variables and constants. Uniqueness of type in the case of variables and constants is thus essential, to permit this identification. It is moreover not clear that much would be gained by allowing such additional forms of overloading (the effect of an overloaded constant can be achieved by a parameterless function). Overloading of type names would lead to unsolvable ambiguities; for example, T(X) could either be a qualified expression or a function call.

The remainder of this section analyzes the identification process and redeclarations. We conclude by comments on the good usage of overloading.

7.5.1 Identification of Overloaded Constructs

When an overloaded identifier appears at a given point of the text, the identifier itself does not provide enough information to identify a unique meaning. In such cases *contextual* information must be used to select the unique meaning, and the occurrence is ambiguous if this information is not sufficient. This problem arises for the following language constructs:

- literals
- aggregates
- calls of parameterless functions
- calls of subprograms with parameters
- calls of subprograms defined in packages or tasks and made visible via a use clause.

By contextual information we mean information derived from the context where these constructs are used, such as the type of a variable to which a literal, an aggregate, or a function is assigned, or the types of the actual parameters of a subprogram call. Whenever this information is not sufficient the programmer must provide explicit qualification (for example by a qualified expression). The extent to which this is required depends upon the particular rules adopted for overloading. We discuss these alternative rules using the following example:

```
function "*"(X, Y : COMPLEX) return COMPLEX;
function "*"(A : INTEGER;  Y : COMPLEX) return COMPLEX;
function "*"(Y : COMPLEX;  A : INTEGER) return COMPLEX;
```

First consider the variable declarations

```
C, D   : COMPLEX;
I, J   : INTEGER;
M, N   : LONG_INTEGER;
```

Expressions including a "*" operator such as

```
C * D
I * J
I * C
M * N
```

are unambiguous since the type of both operands is known and can be used to identify the corresponding operator specification. But now take the following cases:

(a) INTEGER(3*5);

(b) I := 3*5;

(c) C := I*3*D;

(d) C := 3*5*D;

(e) C := 3*(1,5);

Several alternative overloading rules can be considered.

Rule 1:

The types of the parameters must be sufficient for the identification.

This rule requires a single *bottom-up* traversal (from leaves to root) of the trees associated with expressions. It is the rule used in Algol 68. This rule permits the identification of "*" in case (c) but not in the other cases, because (for example) the type of 3 is not determined.

Rule 2:

The types, modes, and names of the parameters must be sufficient for the identification.

This rule would permit the additional identification of subprograms and aggregates with named associations. For the above example, only case (c) is identified.

Rule 3:

The types, modes, and names of the parameters, together with the result type, must be sufficient for the identification. Here the *desired* type derived from the context is first propagated down the tree associated with the expression, then the bottom-up traversal is performed.

As examples of this influence of context: the type of the variable on the left-hand side of an assignment is the desired type for the right-hand side; the type given in a qualified expression is its desired type; the type of a formal parameter of an identified subprogram is the desired type of the corresponding actual parameter. This rule resolves cases (a), (b), (c). It does not resolve cases (d) or (e) since none of the occurrences of "*" can be completely resolved.

In order to explain this we shall use the notation of qualified expressions and express the multiplication in functional form (MUL is used instead of "*"). We use ? to denote an undefined qualification. To assist the eye, contents of widely separated parentheses have been dropped by one line. For example, case (c) can be expressed as

```
C := COMPLEX(                                                    );
             MUL(                                                )
                  INTEGER(                                ),COMPLEX(D)
                           MUL(                    )
                                INTEGER(I),INTEGER(3)
```

On the other hand, case (d) corresponds to

```
C := COMPLEX(                                        );
             MUL(                            )
                  ?(                  ),COMPLEX(D)
                    MUL(?(3), ?(5))
```

Note that since the result type is also taken into account, parameterless functions may be overloaded. For example, EMPTY can be defined for several user defined types. Similarly, functions may also be considered to be attributes of a type on the basis of their result type, not only on the basis of their parameter types: a more intuitive rule.

Rule 4:

The types, modes, and names of the parameters, together with the result type must be sufficient for the identification. The contextual information propagated first top-down and then bottom-up in the expression tree can consist of a set of alternative types.

For this rule, the set of alternative types for a parameter is propagated top-down. These alternative types correspond to the types given in all specifications that match the (set of) desired result type(s). During the subsequent bottom-up propagation it must be possible to identify all operations. This rule resolves case (d). Consider for example the top-down propagation (we qualify an expression by a set of types, or by the type itself if it is unambiguous). This gives:

```
C := COMPLEX(                                            );
             MUL(                                    )
                  S1(                  ),COMPLEX(D)
                    MUL(S2(3),S3(5))
```

where the sets are as follows: **S1 = S2 = S3 = {INTEGER, COMPLEX}**

In the subsequent bottom up propagation this desired type information is sufficient to determine that 3 and 5 are of type INTEGER thus resolving the statement as:

```
C := COMPLEX(                                                    );
             MUL(                                                )
                  INTEGER(                          ), COMPLEX(D)
                          MUL(                  )
                               INTEGER(3),INTEGER(5)
```

Rule 5:

The types, modes, and names of the parameters, together with the result type must be sufficient for the identification. The contextual information is propagated both ways, repeatedly until convergence.

For this rule, the identification is a success if every set of desired types has a unique member at convergence. Otherwise the expression is ambiguous. This rule identifies case (e):

```
C := 3*(1,5);
```

After the first top down pass we have

```
C := COMPLEX(                                  );
             MUL(                          )
                  S1(3), S2(             )
                         (?(1), ?(5))
```

where the sets S1 and S2 are defined as

```
S1 = S2 = {INTEGER, COMPLEX}
```

After the first bottom up pass we are able to identify the literal 3, thus leading to:

```
C := COMPLEX(                                    );
             MUL(                            )
                  !NTEGER(3),S2(             )
                                (?(1),?(5))
```

A second top down pass now permits in succession the identification of the multiplication operation, the qualification COMPLEX of the aggregate, and the types of the components.

```
C := COMPLEX(                                                    );
             MUL(                                              )
                  INTEGER(3),COMPLEX(                        )
                                     (INTEGER(1),INTEGER(5))
```

For practical purposes, only the last two rules would seem to be sufficient. The algorithm implied by the fifth rule could in theory be rather time-consuming. However, in practice most expressions are rather small and convergence towards a definition (or towards ambiguity) should be rather fast in view of the fact that named constants and variables is the usual case.

The major reason for adopting this fifth rule in the Green language, however, is not so much the additional power (since qualification may be used for more readability) but rather the simplicity of its statement. In effect this rule is the only one for which the following statement is true:

> *The types, modes, and names of the parameters together with the result type must be sufficient for the identification.*

The additional details about propagation need not be stated: if it is sufficient, it can be done in this manner or in any other manner.

7.5.2 Redeclaration of Subprograms

The problem of determining whether a subprogram declaration is a redeclaration of a prior subprogram declaration, is closely connected to the previously discussed identification of overloaded subprogram calls.

The rule adopted for overloading uses all available information, including the type of parameters and result, their names, and modes (also order for positional associations). As a consequence a subprogram declaration is a redeclaration only if all calls are ambiguous. If only part of the calls are ambiguous the declaration is an overloading. In any case ambiguous calls can always be detected.

7.5.3 Overloading of Operators

As stated earlier, good usage of overloading can contribute to readability by allowing suggestive identifiers to be used. However, as with any other facility, overloading could be misused. For example, the language definition does not (and cannot) prevent a user from declaring a GET function that actually performs an output operation. In general it cannot prevent the choice of names that give the wrong connotation.

This danger is particularly apparent for operators since the corresponding symbols convey a well defined mathematical meaning. Users are therefore strongly advised to preserve the usual axioms that apply to the predefined operators. The language only permits overloading of = to deliver a result of type BOOLEAN. Similarly, /= is interpreted as the predefined opposite of = and hence cannot be overloaded directly. On the other hand, the same danger exists for other operators, for example the relational operators <, <=, >, >= and the user should exercise extreme care in such cases. Separate overloading of the operator <=, for example, may permit a more efficient implementation than would be derivable by combining separate implementations of < and =.

We believe that the language designer should not forbid an otherwise useful facility, on the grounds that it could be misused in isolated cases. He should never take the attitude of the Newspeak [Or 50] designer:

> *"Don't you see that the whole aim of Newspeak is to narrow the range of thought? In the end we shall make thought-crime impossible, because there will be no words in which to express it."*

Rather, he should always strive to expand the expressive power of the language, while at the same time providing more safety by the consistency of his design.

8. Modules

8.1 Motivation

Modules allow the programmer to group logically related items together. They cover a wide family of uses, ranging from collections of common declarations, to groups of subprograms and encapsulated data types. The Green language provides two forms of modules called *packages* and *tasks*, with similar properties. Tasks are the form of modules used for parallel processing. This discussion of modules is presented essentially in terms of the package form, although the same considerations generally apply to tasks. Tasks are further discussed in Chapter 11.

The ability to package declared entities, such as subprograms, data elements, types, and other modules, provide the basis for a powerful structuring tool for complex programs.

Moreover, modules permit clear delineation of the amount of information accessible to the rest of the program on the one hand, and the information that must remain internal to the module on the other hand. The internal information is hidden and thereby protected from deliberate or inadvertent use by other programmers. This serves not only to localize the effect of errors internal to the module, but also to increase the ease with which one implementation of the module may be replaced by another. Because of this second aspect, modules are also an essential tool for program modularity, supporting information hiding [Pa 71], and program verification.

Facilities for modularization have appeared in many languages. Some of them such as Simula [DNM 69], Clu [Li 74, LSA 77], and Alphard [WLS 76] provide dynamic facilities which may entail extensive run-time overhead.

The facility provided in the Green language is more static, in the spirit of previous solutions offered in the Modula [Wi 76], Euclid [LHLMP 76], Lis [IRHC 74, IF 77] and Mesa [GMS 77] languages. At the same time it retains the best aspects of solutions in earlier languages such as Fortran and Jovial.

The solution provided here combines powerful facilities which nevertheless remain efficiently implementable within the state of the art.

We shall first discuss packages informally by means of examples (section 8.2). We then discuss a number of important technical issues addressed during the design of the language (section 8.3).

8.2 Informal Introduction to Packages

Packages, as provided in the Green language, are a major tool for program modularity. We recognize three broad forms of modularization that can be achieved by different uses of modules:

(1) *Named collection of declarations*:

Logically related variables, constants, and types to be used in other program units.

(2) *Groups of related subprograms*:

Logically related functions and procedures which share internal own data, types, and subprograms. This form of module corresponds to what is commonly called a *package*. By extension we use the same term for all three forms.

(3) *Encapsulated data types*:

Definition of new types and associated operations in such a way that the user does not know (or care!) how the operations are implemented.

The essential difference between these three forms is the amount of information hiding provided by the module. The hiding can be envisioned as a *wall* surrounding the module, thereby separating the enclosed declarations from the rest of the program. One may then imagine a *window* in the wall, through which some or all of the declarations (depending on the size of the window) are exposed. Access to the exposed declarations can be achieved either by selected components (dot notation) or with a *use* clause. For the three broad forms of modules we have:

(1) *Named collection of declarations*:

The module exposes all of its declarations (all declarations can be seen through the window).

(2) *Groups of related subprograms*:

The module exposes the declarations of the externally accessible subprograms (only these can be seen through the window) but hides the declaration of internal variables and other local entities.

(3) *Encapsulated data types*:

The module exposes only the type name and the declarations of applicable operations, but hides all details of structure, representation and implementation of the operations, as well as any variables needed for communication among the operations. Several related types may be encapsulated in the same module.

There is no critical linguistic difference between these three categories, and programmers may use any intermediate degree of information hiding. However, to present the ideas simply we discuss the three broad forms separately, with appropriate examples.

8.2.1 Named Collection of Declarations

The most traditional use of named collections of variables is as communication areas, where several program units share access to a single collection. As an example, in a simple graphic application, the following module declaration may be provided:

```
package PLOTTING_DATA is
  PEN_UP : BOOLEAN;

  CONVERSION_FACTOR,
  X_OFFSET, Y_OFFSET,
  X_MIN, X_MAX,
  Y_MIN, Y_MAX : REAL;

  X_VALUE, Y_VALUE : array(1 .. 500) of REAL;
end PLOTTING_DATA;
```

The elaboration of this module consists of the elaboration of its constituent variable declarations. Elaboration takes place in the context where the package declaration appears textually. Thus in terms of the lifetime of the constituent variables (PEN_UP, ..., Y_VALUE), everything happens as if their declarations were inserted in the place of the declaration of the package PLOTTING_DATA.

However, when a package declaration is elaborated, the constituent variables are not yet directly accessible. In any context where the module declaration can be seen it is possible to acquire access to these variables by dot notation. For example we could write statements such as

```
PLOTTING_DATA.PEN_UP := TRUE;
PLOTTING_DATA.X_VALUE(10) := PLOTTING_DATA.X_MIN;
```

It is also possible to acquire access to these variables by means of a use clause of the form

```
use PLOTTING_DATA;
```

The effect of a use clause is to make a set of names known locally. The names and their meanings are defined in the designated module. In this fashion, with a use clause, one may selectively acquire access to an entire group of declarations. As an example, the previous statements can be rewritten as

```
declare
  use PLOTTING_DATA;
begin
  PEN_UP := TRUE;
  X_VALUE (10) := X_MIN;
end;
```

This simple form of package corresponds almost exactly to the notion of named common in Fortran. There are however three crucial differences between this use of packages and Fortran named common blocks:

(1) A moduie can be declared in any nested block or program unit, so long as its declaration is then visible to all program units requiring access to the module. By contrast, in Fortran all named common blocks are effectively global to the main program.

(2) Storage reservation for a module (as a consequence of (1) above) may be performed at scope entry time for a module that is not global to the entire program. By contrast, the storage space for a Fortran named common block is always reserved throughout the entire program execution.

(3) The contents of a module is defined only once, in the context of its declaration. Within the scope of the declaration, it is then possible to acquire access to that entire set of entities, in as many program units as necessary, merely by mentioning the name of the given module in a use clause (even in the case of separately compiled units). For Fortran named common blocks on the other hand, the specification must be replicated in its entirety in each context where access to the constituents is required. The need to replicate information in this fashion is generally recognized as a violation of the principles of modularity, as an inconvenience and as a serious source of errors.

A similar class of modules is for groups of constant declarations, for example

```
package METRIC_CONVERSIONS is
   CM_PER_INCH  : constant REAL := 2.54;
   CM_PER_FOOT  : constant REAL := 30.48;
   CM_PER_YARD  : constant REAL := 91.44;
   KM_PER_MILE  : constant REAL := 1.609344;
end METRIC_CONVERSIONS;
```

More generally, in a typed language, groups of declarations are likely to include logically related types, along with constants, and variable declarations as shown in the next example

```
package WORK_DATA is
   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   type DURATION   is delta 0.01 range 0.0 .. 24.0;
   type TIME_TABLE is array (MON .. SUN) of DURATION;

   WORK_HOURS    : TIME_TABLE;
   NORMAL_HOURS  : constant TIME_TABLE :=
                   (MON .. THU => 8.25, FRI => 7.0, SAT | SUN => 0.0);
end WORK_DATA;
```

In all three examples we achieve the same effect: the elaboration of the module *creates* the corresponding entities (whether a constant, a variable, or a type), but they are not yet externally visible. Accessibility is only obtained by dot notation or by a use clause. In a unit with a use clause for WORK_DATA, it is possible to declare variables of type DURATION, and to update the array WORK_HOURS, or to read the constants of NORMAL_HOURS. Thus we may have:

```
declare
   use WORK_DATA;
   TODAY  : DAY;
   HOURS  : DURATION;
begin
   -- compute HOURS and TODAY
   if HOURS > NORMAL_HOURS (TODAY) then
      HOURS := HOURS + 2.0 * (NORMAL_HOURS(TODAY) - HOURS);
   end if;
   WORK_HOURS (TODAY) := HOURS;
end;
```

8.2.2 Groups of Related Subprograms

The second major use of modules is for the creation of groups of related subprograms. For example, it may be desirable to make a package of the mathematical routines (SIN, COS, LOG, EXP, ...) for the reason that a user needing one of them is very likely to need the others too. Moreover, the routines may share common subprograms (sometimes elaborate) which should not be accessible to the user.

Declaring such functions within a module MATH_LIB is preferable to having them as predefined functions in the standard environment. Thus, a user who is not dealing with numerical computations does not have to import MATH_LIB, and his *name space* will not be congested by names that are useless to him.

We next consider the example of a package of random number generators, since it will enable us to point out other important possibilities. The structure of this package is already more elaborate than the form used for collections of common declarations. It is made of two parts which together define the module. The first part is the module specification:

```
package RANDOM is
   -- visible part : declarations of sampling functions
end RANDOM;
```

This describes the visible part of the package, that is, the declarations that become directly accessible in a context containing a use clause for RANDOM. In this case, the user interface contains the declarations of the sampling functions DRAW, RAND_INT and UNIFORM. It also includes the procedure SET_SEED used to initialize the *seed* of the random number generator, and the exception ERROR. Consequently the user may decide to provide a local handler for ERROR.

The second part of the module is the module body. This is the hidden part of the module; none of the entities contained therein are accessible outside the module (both use clauses and selected components only give access to items of the visible part). The structure of the package body is as follows:

```
package body RANDOM is
   -- hidden data and subprogram bodies
begin
   -- initialization part
end RANDOM;
```

The two variables SEED and CO_SEED (like all variables of a module that are not local to inner subprograms) are *own* variables. They are initialized by the initialization part of the module and retain their value between calls of the sampling functions. In languages which do not have this facility for hiding own variables, the seed must either be global, or be passed as a parameter to the sampling procedures; in either case it is therefore *known* outside the module.

The package body also contains the bodies of the sampling procedures, and that of the procedure CONGRUENCE for generating the next random number. This procedure is purely local and is not part of the user interface (it is not declared in the visible part). Hence it can only be called within the module body of RANDOM by the sampling functions.

```
package RANDOM is
   procedure SET_SEED (START : INTEGER);

   procedure DRAW (CHANCES : REAL range 0.0 .. 1.0) return BOOLEAN;
   procedure RAND_INT (LOW, HIGH  : INTEGER) return INTEGER;
   procedure UNIFORM  (LOW, HIGH  : REAL) return REAL;

   ERROR : exception;
end RANDOM;

package body RANDOM is
   SEED         : INTEGER;
   CO_SEED      : REAL;

   procedure CONGRUENCE is
      LAMBDA : constant INTEGER := 8#12_571_342_215;
   begin
      SEED      := INTEGER((LONG_INTEGER(SEED)*LONG_INTEGER(LAMBDA)) mod 2**32);
      CO_SEED   := REAL(SEED) / REAL (2**32);
   end;

   procedure SET_SEED(START : INTEGER) is
   begin
      SEED  := START;
      CONGRUENCE;
   end;

   procedure DRAW(CHANCES : REAL range 0.0 .. 1.0) return BOOLEAN is
   begin
      CONGRUENCE;
      return CO_SEED < CHANCES;
   end;

   procedure RAND_INT(LOW, HIGH : INTEGER) return INTEGER is
   begin
      if LOW > HIGH then raise ERROR; end if;
      CONGRUENCE;
      return LOW + INTEGER(CO_SEED * REAL(HIGH - LOW + 1));
   end;

   procedure UNIFORM(LOW, HIGH : REAL) return REAL is
   begin
      if LOW > HIGH then raise ERROR; end if;
      CONGRUENCE;
      return LOW + CO_SEED*(HIGH - LOW);
   end;
begin
   -- package initialization :
   SET_SEED(1);
end RANDOM;
```

The separation of the two parts of a module (the module specification and the module body) is very clear. In general the two parts need not be textually contiguous and they may even be compiled separately. In this way the separately compiled module body is not only protected but *physically* hidden.

In a language with type declarations such as the Green language, packages developed for user applications are likely to contain not only procedures but also type declarations. As an example, consider the package RATIONAL_NUMBERS.

```
package RATIONAL_NUMBERS is
   type RATIONAL is
      record
         NUMERATOR     : INTEGER;
         DENOMINATOR   : INTEGER range 1 .. INTEGER'LAST;
      end record;

   function "="  (X,Y : RATIONAL) return BOOLEAN;
   function "+"  (X,Y : RATIONAL) return RATIONAL;
   function "*"  (X,Y : RATIONAL) return RATIONAL;
end;

package body RATIONAL_NUMBERS is

   procedure SAME_DENOMINATOR (X,Y : in out RATIONAL) is
   begin
      -- reduces X and Y to the same denominator
   end;

   function "=" (X,Y : RATIONAL) return BOOLEAN is
      U,V : RATIONAL;
   begin
      U := X;
      V := Y;
      SAME_DENOMINATOR (U,V);
      return (U.NUMERATOR = V.NUMERATOR);
   end "=";

   function "+"  (X,Y : RATIONAL) return RATIONAL is    ... end "+";
   function "*"  (X,Y : RATIONAL) return RATIONAL is    ... end "*";

end RATIONAL_NUMBERS;
```

The type RATIONAL is declared within the visible part of the package. In a context containing a use clause for RATIONAL_NUMBERS, it is possible to declare variables of type RATIONAL and apply the operators "=", "+", and "*" to them. It is also possible to operate directly on the components of a rational number and to construct rational values as record aggregates.

As in the previous example, the bodies of these functions are provided in the module body, which also declares the local procedure SAME_DENOMINATOR. This procedure modifies the denominators (and numerators) of its arguments. By encapsulation, usage has been confined to the bodies of the three public functions, where proper precautions can be taken (for example, copying the arguments X and Y of "=" in the local variables U and V before calling SAME_DENOMINATOR).

8.2.3 Encapsulated Data Types

In the previous package examples, the hiding of declarations and subprogram bodies within the module body ensured that no outside program unit could affect these local entities. In this simple model, entities were either public (if declared in the visible part) or totally hidden (if declared in the module body).

Encapsulated data types correspond to a situation in which we want the name of a type to be public, but where the knowledge of its internal properties is to be available only to the subprogram bodies contained in the module body.

This encapsulation is achieved by declaring the type name within the visible part (since the type name should be available to users of the module), but at the same time specifying the type to be *private*. Its full definition is then provided in a private part following the visible part.

As an example of encapsulated data type, consider the following skeleton of an input output package declared as follows:

```
package SIMPLE_INPUT_OUTPUT is

   type FILE_NAME is private;
   NO_FILE : constant FILE_NAME;
   procedure CREATE return FILE_NAME;
   procedure READ   (ELEM : out  INTEGER; F : in FILE_NAME);
   procedure WRITE  (ELEM : in   INTEGER; F : in FILE_NAME);

private
   type FILE_NAME is new INTEGER 0 .. 50;
   NO_FILE : constant FILE_NAME := 0;
end SIMPLE_INPUT_OUTPUT;
```

In the visible part given above, the type FILE_NAME is declared as private. External to the module it is possible to declare variables of the type, but the properties of objects of this type are kept private. Hence the only thing a user can do with file names is to assign them to other file name variables or to pass them as parameters (mainly to the procedures READ and WRITE) once they are obtained by calling the procedure CREATE.

The full definition of the private type FILE_NAME and that of the private constant NO_FILE are given in the private part (the declarative part following the reserved word **private**). A module body for the above package is sketched as follows:

```
package body SIMPLE_INPUT_OUTPUT is
   type FILE_DESCRIPTOR is record    ... end record;
   DIRECTORY : array (FILE_NAME) of FILE_DESCRIPTOR;
   ...
   procedure CREATE return FILE_NAME is    ... end;
   procedure READ   (ELEM : out  INTEGER; F : in FILE_NAME) is    ... end;
   procedure WRITE  (ELEM : in   INTEGER; F : in FILE_NAME) is    ... end;
begin
   ...
end SIMPLE_INPUT_OUTPUT;
```

Within the body, file names are integers indexing an internal directory declared as an array. However, an external user of the module cannot, for instance, do any arithmetic on file names.

With the above definition of the type FILE_NAME, the user still has the possibility of assigning file names and also of comparing them for equality and inequality. These possibilities are even denied in the following variation of the previous package:

```
package SAFE_IO is
  restricted type FILE_NAME is private;

  procedure CREATE  (F : in out FILE_NAME);
  procedure READ    (ELEM : out  INTEGER; F : in FILE_NAME);
  procedure WRITE   (ELEM : in   INTEGER; F : in FILE_NAME);
private
  type FILE_NAME is
    record
      CONTENT : INTEGER := 0;
    end record;
end SAFE_IO;
```

A user of this module can only:

- declare variables of type FILE_NAME; the use of a record type containing initial values provides implicit initialization of such variables.
- pass these objects to the operations supplied by the package SAFE_IO.
- pass them as parameters of other procedures. For example, it is possible to write the following procedure

```
procedure TRANSFER_ITEM (F,G : in FILE_NAME) is
  E : ELEM;
begin
  READ   (E,F);
  WRITE  (E,G);
end;
```

Since neither assignment nor comparison of file names are possible, it is unnecessary to define a *null* file in this formulation. Moreover, CREATE can no longer be a value-returning procedure since its result could not be assigned to the destination variable. Note also that the parameter mode for CREATE has been changed to **in out** thus permitting us to check whether it is overwriting an existing file name.

For the more classical examples of encapsulated data types (from the current literature), the reader is referred to chapter 13 of this document (a generic definition of the type queue) and to section 12.3 of the reference manual (a generic definition of the type stack).

8.3 Technical Issues

The design of modules involves nearly all aspects of the language. The most significant in this context are

- Visibility control and information hiding
- Relation to separate compilation
- Instantiation and initialization of modules
- Access to the properties of types defined within modules
- Instantiation and Initialization of objects of private types.

Other interactions will be discussed in the chapters on program structure and visibility, on parallel processing, on separate compilation, and on generic units.

8.3.1 *Visibility Control and Information Hiding*

The scope rules of traditional block structure are inadequate for the reliable construction of large programs. Declarations in nested blocks allow entities of outer blocks to be accessed by all inner blocks, whereas one needs more precise control over the visibility of declarations. Another severe limitation of block structure is the inability to declare a variable (or any other name) accessed by a limited set of subprograms without making the variable, at least potentially, accessible for other subprograms also.

Modules give the programmer precisely this kind of control. The corresponding visibility rules are discussed in the chapter on program structure. In this chapter we concentrate on the characteristics of modules that are essential for visibility control and information hiding.

When defining a module, the visible part states which declarations are potentially accessible outside the module. This identifies the window in the wall surrounding the module mentioned above.

Any other program unit within the scope of the module can potentially access the visible part, but does not do so automatically. In order to get actual access, either a use clause must be provided, or names in the form of selected components (dot notation) must be written.

Thus, access to the identifiers declared in the visible part is selective. Names declared in the visible part of a module do not automatically pollute the name space of the rest of the program. Access to the identifiers declared in the module body is even more controlled. Access is available only within the module body, in particular to the bodies of the subprograms declared in the visible part.

The other essential characteristic of modules in this language is the textual separation of the interface.

The interface of a module consists of all relevant declarations for a user of the module. In the solution adopted for the Green language, all these declarations are textually separated from the rest of the text. They form what is called the visible part of the module. This textual separation is a significant advantage for readability and for information hiding.

Other languages such as Euclid and Modula have used a solution involving an export list mentioning all identifiers constituting the module interface. This means that either all meanings of an overloaded identifier are exported, or none of them. Moreover it also means that in order to know the properties of these identifiers, the human reader must scan through the entire text of the module. This is a tedious operation and is, in addition, a breach of information hiding principles, since it involves reading parts of the text which should be of no concern (and may not even be available) to the user.

8.3.2 Influence of Separate Compilation

The essential role of modules is for logical modularity. However they also have a useful role in physical modularity in terms of separate compilation. These two criteria for program modularization lead to slightly different requirements, and the design of modules takes both into account. Both require modules to have a clearly defined interface, but physical modularity requires an interface containing more information than for logical modularity.

Extra information is needed to allow the compiler to deal with variables of private types seen from a separately defined module. For this reason the private part belongs to the physical interface.

From the point of view of logical modularity, the physical interface is irrelevant to all other modules, since they should not be concerned with the physical representation of the encapsulated types. But in order to permit separate compilation, it is necessary for any module specification to provide sufficient information to allow the compiler to handle declarations of variables of an encapsulated type.

The difference essentially concerns storage allocation. A type declaration implicitly defines the amount of storage needed for each variable of the type. A module defining a type must therefore mention the entire structure of the type so that the appropriate amount of space can be reserved. This information (and also any representation specification for the type) must be part of the physical interface of the module. Otherwise any change will require that all dependent modules be recompiled.

To summarize, the logical interface corresponds to the visible part; the physical interface corresponds to the complete module specification, that is, to both the visible and the private part.

As long as a module specification is not changed, the module body that implements this specification can be defined and redefined without affecting other units using this specification as interface to the module. Hence it is possible to compile a module body separately from its module specification.

8.3.3 Instantiation and Initialization of Modules

Each package declaration results in a single instance of the corresponding package. This instance is created when the package is elaborated. As a consequence, the space needed for the objects declared in the package (both in the package specification and in the package body) is allocated. Whenever such an object declaration specifies an initialization, it is performed.

More elaborate initializations, which cannot be specified by expressions but require the execution of statements, can be included in the sequence of statements following the (optional) reserved word **begin** in the package body. The execution of this (optional) sequence of statements completes the elaboration of the package. An exception handler provided at the end of these statements applies for exceptions raised during the elaboration of the package declaration.

Multiple instances of a module can be obtained if the module is generic. In this case the module specification includes a generic clause and individual instances are created by generic instantiation.

Elaboration of a task body is performed differently. Elaboration is performed when an initiate statement for the task is executed. This is discussed further in the chapter on tasking.

8.3.4 *Note on Visibility*

If a use clause is provided within a given program unit, it opens the visibility of the visible part of each named module in a non-transitive manner. Thus, if a use clause

```
use M;
```

is given in the visible part of a module D, it does not mean that units containing a use clause

```
use D;
```

will also see M. When this kind of transitivity is desired it can be achieved explicitly by declarations, in particular by renaming declarations.

Consider for example

```
package M is
   type T is private;
   procedure P(X : T);
   ...
   E : exception;
end M;
```

Suppose now that a package D defines additional operations for the type T in terms of the operations supplied by M, and suppose we want to make T, P, and E available to all users of the package D without an explicit **use** M clause. This can be achieved as follows:

```
package D is
   subtype T is M.T;
   procedure P renames M.P;
   -- additional operations defined by D
   E : exception renames M.E;
end D;
```

Note that we could also have declared the type T by a derived type definition

```
type T is new M.T;
```

This latter form could be used if we wanted to forbid the use of operations defined by another package for objects of type M.T at the same time as those defined by the package D.

8.3.5 Access to the Properties of Types Defined Within Modules

It is important to define which of the properties of a type declared in the visible part of a module are accessible outside the module (for example within another unit mentioning the module in its use clause)? The answer to this question is simple: the only available properties are those declared in the visible part.

As a first case, consider a type declaration of the usual form, say a record type declaration. If such a declaration appears in the visible part of a module, this type is available without restrictions to outside units. In particular these can declare variables and carry out operations (such as component selection, etc.) in full knowledge of the data structure specified by the type.

For a type declared as *private*, on the other hand, the visible part gives only the type name and the specification of the subprograms applicable to objects of this type. These subprograms are said to be *attributes* of this type. They are the only operations applicable to objects of the type apart from assignment and comparison for equality or inequality (unless the private type is *restricted*).

The separation of inheritance properties by these rules is elementary and its logical basis is clear. More complicated rules have been investigated, but all involve elaborations of the language in an area which is a subject of current research.

Since these attributes are the only properties of a private type that are accessible outside the module, they are also the only properties that can be inherited in a derived type definition. As an example, consider the following declaration of the type MY_FILE:

```
declare
   use SIMPLE_INPUT_OUTPUT;
   type MY_FILE is new FILE_NAME;
   ...
begin
   ...
end;
```

Within the block, the only operations available on objects of type MY_FILE are assignment, comparison, the function CREATE, and the procedures READ and WRITE.

Conceptually we may view predefined types such as INTEGER and FLOAT as being types thus declared in predefined packages, along with their attributes. Inheritance of their properties in declarations such as

```
type MY_REAL is new FLOAT;
```

is hence only the result of the application of the general rule.

Within a module body the characteristics of a private type are known as if the type were not private. For example if the type is a record type, its components can be denoted with the usual syntax of selected components. Some precautions must be taken when one of the visible attributes of the type is defined in terms of an existing attribute with the same name. As an example consider the skeleton of the package KEY_MANAGER given in the Reference Manual (section 7.4):

```
package KEY_MANAGER is
   type KEY is private;
   ...
   function "<"(X,Y : KEY) return BOOLEAN;
private
   type KEY is new INTEGER range 0 .. INTEGER'LAST;
end;

package body KEY_MANAGER is
   ...
   function "<"(X,Y : KEY) return BOOLEAN is
   begin
      return INTEGER(X) < INTEGER(Y);
   end "<";
end KEY_MANAGER;
```

Within the package body the definition of the derived type KEY is known. The operation "<" declared in the visible part is a (perfectly legal) redeclaration of the operation "<" inherited from the type INTEGER. Thus, with the declarations

```
U, V : KEY;
```

within the body of the package, the relation

```
U = V
```

refers to the operation "=" inherited from integer, whereas the relation

```
U < V
```

refers to the operation "<" defined within the module itself (in this case, of course, it does not matter since this redefinition is equivalent to the inherited operation). It should be noted that within the body of the function "<" itself, the relation

```
X < Y
```

would be a recursive call of the function "<". In order to obtain the operation defined on integers, a qualification must be used. For example

```
INTEGER(X) < INTEGER(Y)
```

invokes the operation < defined on integers.

8.3.6 Instantiation and Initialization of Objects of Private Types

The elaboration of an object declaration results in a static reservation of space for the corresponding object, whether the type of the object is private or not.

The initialization of an object of a private type may be achieved by assigning a private constant or a function returning a result of the type. However, there are cases where we want the *representation* of an object of a private type to satisfy some invariant as soon as the object is created, although a complete initialization would be redundant. This is achieved by means of initialization of record components. Consider the following package declaration:

```
package ALL_ABOUT_STACKS is
   restricted type STACK is private;
   procedure PUSH  (E : in    ELEMENT; S : in out STACK);
   procedure POP   (E : out   ELEMENT; S : in out STACK);
private
   type STACK is
      record
         TOP          : INTEGER range 0 .. 1000 := 0;
         SPACE        : array(1 .. 1000) of ELEMENT;
      end record;
end;
```

For any declaration of an object of type STACK, the component TOP is initialized to zero. Thus, the stack invariants are satisfied as soon as a stack is elaborated (another example was shown in section 8.2.3 above, with the initialization of file names in the package SAFE_IO).

This initialization capability is clearly limited, but should be sufficient in most cases. When more complicated forms of initialization are needed, the visible part of the module must provide the specification of an appropriate procedure performing the initialization. This procedure must then be called by the programmer whenever declaring a variable of the private type, prior to any other operation.

8.4 Conclusion and Summary

Modularity seems to have emerged as one of the overriding concerns in contemporary programming. Unfortunately, this one simple word encompasses a number of desirable but not necessarily compatible objectives, which gives rise to as many different interpretations as there are priorities among the various objectives. It comes as no surprise, therefore, to find that there exists no real consensus as to how this elusive quality is to be achieved in practice, let alone about how to provide appropriate support for modularization within a high order programming language.

Curiously enough in such a situation, the Steelman requirements are quite specific as to the kinds of facilities that shall be provided for this purpose. Since we strongly concur with the requirements as stated, many alternatives that have been proposed elsewhere have not been discussed in this chapter. For a more extensive examination of the different linguistic approaches, the reader is referred to [GK 1977].

A straightforward approach was taken for the module facility in the Green language. Modules provide an ability to package information. When defining a module the programmer simply states the visible information and provides the module implementation as a separate text (its body). Access to information of a module body is not (directly) possible outside the module body. Thus, modules support information hiding as well as control of visibility.

The module facility is central to the definition of encapsulated data types; it provides complete control over the available operations for such types. Finally, modules can be parameterized by generic clauses, separately compiled, and entered in libraries.

All of these aspects are in many respects fundamental for program development. Modules are expected to be used to construct libraries containing common pools of data and types, application packages, and complete systems.

9. General Program Structure and Visibility

9.1 Introduction

Central to the definition of the Green language has been a combined study of the general program structure, of the rules defining the visibility of identifiers at various points of a program, and of the facilities offered for separate compilation.

A major goal in this design was to give the programmer precise control over his *name space* : the set of names that he may define and use. It is important to be able to introduce new names without having to bother about possible conflict with preexisting names. This requires an ability to control the inheritance of names from other contexts. As mentioned in a previous chapter, the notion of module is essential to achieve this kind of control.

Another goal was to provide the same visibility rules for all program units (subprograms, modules and blocks) whether they are separately compiled or not.

The subjects of general program structure and of visibility rules are connected in many ways, in particular because of the possibility of nesting program units. Both subjects are also related to the issue of separate compilation. This chapter will discuss program structure and visibility in that order. The subject of separate compilation is treated in the next chapter.

9.2 Program Structure

The overall structure of a Green program text (that is, a compilation unit) is similar to that of an Algol 60 or Pascal text. It appears as a nested structure of program units, that are subprograms, modules, and blocks.

Nesting is achieved through declarative parts. A declarative part may contain subprogram and module bodies. These may in turn contain a declarative part. Nesting is also achieved by blocks in sequences of statements.

A key question in the definition of the general program structure is that of the purpose of nesting. Clearly, nesting has been used in Algol 60 and Pascal in relation to scope. In such languages, two units are written in the context of the same declarative part if they are to share the visibility of some common outer objects.

Is this, however, the only purpose of nesting? If it were, a logical conclusion would be the systematic unnesting of units that do not share any common visibility.

We consider this view to be too extreme. Units that do not have any visibility dependence may nevertheless be maintained together in a nested text structure for benefit of the logical exposition of the program. An analogy may be made with an encyclopedia, where the materials are organized into subjects and sub-subjects. It is the knowledge of this organization that permits an easy retrieval of a given subject.

Systematic unnesting of units that do not share any common visibility would produce a sequence of small units not unlike a sequence of Fortran subprograms. Finding a given unit in such a sequence is difficult unless aided by a dictionary. Reading the program may also be difficult since the structure of the text does not reflect the logical organization and the logical connections.

For these reasons, the possibility of nesting units has been retained. Moreover, provisions have been made to control, and possibly to restrict, the visibility of nested units.

In a declarative part we may thus have a sequence of declarations followed by a sequence of bodies of nested program units. In particular, the sequence of declarations may include the specification of nested subprograms and modules. In general it is possible to provide the definition of subprograms and modules in two textually distinct parts:

(a) the specification, which indicates the logical interface (between definition and use) of the unit at hand

(b) the body, which describes the particular realization for the specification.

This possibility has far reaching implications, in that it provides a single basis for achieving several different objectives, notably textual clarity, abstraction, and separate compilation. We first illustrate this ability in the case of a procedure. For instance, one might write the specification

```
procedure PUSH (E : in ELEMENT; S : in out STACK);
```

This specification contains the name of the procedure and the specification of the mode and type of its formal parameters. Such a declaration is sufficient to specify the *interface* of PUSH, both syntactically and semantically, at least with regard to type checking. From this point of view the specification conveys all one needs to know in order to make use of (i.e. call) the procedure PUSH. The specification could be augmented by comments specifying pre-conditions and post-conditions and any exceptions possibly raised by PUSH.

Obviously however, this formulation of PUSH is incomplete in that it does not define an actual implementation of the procedure. The latter is provided as a procedure body:

```
procedure PUSH (E : in ELEMENT; S : in out STACK) is
begin
   if S.INDEX = S.SIZE then
      raise STACK_OVERFLOW;
   else
      S.INDEX := S.INDEX + 1;
      S.SPACE (S.INDEX) := E;
   end if;
end PUSH;
```

These two constructs, the declaration and the body, jointly form the definition of the procedure. For reasons of syntactic convenience the procedure declaration may be omitted if the body appears in the same declarative part. For reasons of readability the specification is repeated in the body.

A similar partition is provided for modules. A module specification provides the interface to the user, that is, it indicates whether the module is a package or a task and states the properties of its visible part. For example the specification of a SIMPLE_INPUT_OUTPUT package is provided as follows:

```
package SIMPLE_INPUT_OUTPUT is

   restricted type FILE_NAME is private;

   procedure CREATE (FILE : out FILE_NAME);
   procedure READ   (ELEM : out  INTEGER; F : in FILE_NAME);
   procedure WRITE  (ELEM : in   INTEGER; F : in FILE_NAME);

private

   type FILE_NAME is new INTEGER range 0 .. 50;

end SIMPLE_INPUT_OUTPUT;
```

In this example, the specification of SIMPLE_INPUT_OUTPUT provides the user with the specification of the name of the type FILE_NAME and also with the specification of the associated procedures CREATE, READ and WRITE. This constitutes the logical interface of the package.

The module implementation is always provided as a textually distinct module body as shown in the sketch below:

```
package body SIMPLE_INPUT_OUTPUT is

   type FILE_DESCRIPTOR is
      record
         -- components of each file descriptor
      end record;

   DIRECTORY : array (FILE_NAME) of FILE_DESCRIPTOR;

   -- other local constants, variables and subprograms

   procedure CREATE (FILE   : out  FILE_NAME) is ... end CREATE;
   procedure READ   (ELEM   : out  INTEGER; F : in FILE_NAME) is ... end READ;
   procedure WRITE  (ELEM   : in   INTEGER; F : in FILE_NAME) is ... end WRITE;
end SIMPLE_INPUT_OUTPUT;
```

As in the case of procedures, the module specification and the module body jointly define the module in question. For pragmatic reasons (a module specification is generally much larger than a procedure specification), the module body does not repeat the information contained in the module specification.

9.3 Scope and Visibility Rules

The visibility rules provided in the Green language combine classical inheritance rules with the ability to control explicitly the set of names that can be accessed within a given program context. This ability follows from the naming conventions, the facilities offered by modules and the visibility restrictions. A renaming capability is also provided.

We first discuss the basic scope model, then the naming conventions, use clauses, visibility restrictions, and renaming. Finally we discuss the rule of linear elaboration of declarations which has been adopted for reasons of readability.

9.3.1 Basic Scope Model

The search for simple and uniform scope rules has led to the adoption of an almost traditional approach: within a program unit, identifiers declared in outer contexts are automatically visible in inner nested contexts unless an explicit visibility restriction is given.

The word *context*, as used here, generally corresponds to a subprogram or block containing a declarative part, or to a record type definition. Thus the basic rule is essentially that of Algol 60. The only departure is that the convention is applicable within a restricted program unit rather than throughout an entire program.

The fundamental reason for selecting this liberal approach is the pragmatic assumption that names declared together are normally meant to be used together. Consider, for instance, the skeleton

```
procedure P is
   type T is ...;              -- type declaration
   V : T;                      -- variable declaration
   procedure Q;                -- procedure declaration

   ...

   procedure Q is
      ...
   begin
      ...
   end Q;
begin
   ...
end P;
```

Presumably the names T, V and Q are defined in the same context (the declarative part of P) because they are intended to be used together, here in the sequence of statements of P. Extending this reasoning to inner contexts means, for instance, that the names T, V, and possibly Q are also visible within the body of Q, so that the body may be directly defined in terms of these names. This suggests the assumption that entities declared in the same context have mutually dependent definitions, unless explicitly stated otherwise.

One alternative considered was to designate certain syntactic categories such as procedures and modules as being always *closed*, and then to require some form of explicit *import directive* in order to make externally declared names visible within such contexts. This was ultimately deemed unacceptable because it would lead to clutter and to long name lists in many common cases.

The following example illustrates the unnecessary redundancy of the directive "*sees* T, C, L", where the procedures P_1 through P_N are obviously meant to work with T, C and L.

```
-- the following is not a Green text

package D is
   type T is ...;
   C : constant T := ...;

   procedure P_1(...);
   procedure P_2(...);
   ...
   procedure P_N(...);
end D;

package body D is
   L : T;

   -- note: "sees T, C, L" is not legal in Green

   procedure P_1(...) sees T, C, L is    ...    end P_1;
   procedure P_2(...) sees T, C, L is    ...    end P_2;
   ...
   procedure P_N(...) sees T, C, L is    ...    end P_N;
end D;
```

Early experience with the Euclid language, where such an approach was taken, has shown that the danger of long name lists is not to be underestimated. Because of transitivity, Euclid import lists can get very long. The danger is then that programmers may tend to use standard import lists in fear of omitting something (as is often done for Fortran common lists). In any case, long name lists are usually skipped when reading and this defeats their very purpose. The classical argument developed by Dijkstra [Di 72], about our inability to deal with a large number of entities at the same time, also applies to long, and therefore unstructured name lists.

The only way to avoid this form of text clutter is to make automatic inheritance the default rule. The argument is that the textual embedding of declarations is already a strong indication of potential dependencies. The systematic inclusion of additional import directives does not usually provide much information that may usefully be exploited by the translator and it is likely to distract the readers (and writers) of programs.

More significantly, we came to the conclusion that whether a given syntactic category should be an *open* scope or a *restricted* (closed) scope is a highly subjective question. The answer may vary from one usage to another, depending on the size of a particular program unit, the depth to which it is nested, the probability of subsequent recompilation, and so on.

It seems clear, therefore, that the syntax of the language should not arbitrarily impose a decision in this regard. For this reason we have adopted the following approach:

- all syntactic constructs that introduce declarations normally inherit the identifiers of outer contexts.
- Specific subprograms and modules may optionally be specified as being restricted program units, i.e. units with a restricted visibility of outer contexts.

9.3.2 Naming conventions

Since classical inheritance of identifiers from outer contexts is the default rule, redeclaration of identifiers is possible, with the effect of hiding the outer definitions within the inner context.

Some of the difficulties existing in identifier redeclarations disappear if the names of the corresponding entities can be written as *selected components*, i.e. using the dot notation. Consider, for example, a type T declared in a procedure P, and assume the identifier T redeclared in an inner context. The type name can still be written as P.T in the inner context (exploiting the fact that the identifier P is visible there) and may thus be used in qualified expressions and so forth, if the need arises.

Naming by selected components is also the general rule used for denoting identifiers declared in packages and tasks outside of their own specification and body. Thus an identifier I declared in the visible part of a module D is denoted by the selected component D.I outside the module, in spite of the fact that I is not directly visible there.

As an additional syntactic convenience a *use clause*, mentioning names of modules, may appear at the start of a declarative part. In the absence of any conflicting identifier, its effect is to permit designation of an identifier of the visible part directly, as if the identifier were declared at the place of the module specification containing it. For example, in a context including the use clause

```
use D, E;
```

the identifier I is an acceptable abbreviation for D.I provided that it is declared in D and is not hidden by an intervening redeclaration of I, and provided also that the module E does not contain an identifier I in its visible part. In all cases of conflict, the name must be given in full as a selected component.

These rules are illustrated by the following example:

```
package D is
   T, U, V : BOOLEAN;
end D;
procedure P is

   package E is
      B, W, V  : INTEGER;
   end E;

   procedure Q is
      T,X : REAL;
   begin
      ...
      declare
         use D, E;
      begin
         -- the name T   means Q.T, not D.T
         -- the name U   means D.U
         -- the name B   means E.B
         -- the name W   means E.W
         -- the name X   means Q.X
         -- the name V   is illegal : it must be written either D.V or E.V
         ...
      end;
   end Q;
begin
   ...
end P;
```

In deciding which names are visible within the sequence of statements of the block we apply the following rule:

(a) First we inherit the names declared in outer contexts and not redefined. Thus we inherit the names D and P, the names E and Q introduced within P, and the names T and X introduced within Q.

(b) Then we consider the names that may additionally be introduced by use clauses. In the above example we only have to consider the names that are in the visible parts of D and E. We retain names that appear in only one of these modules and that do not conflict with a name introduced in the step (a). Hence the names additionally introduced are U, B, W. Nested use clauses would be treated in the same way.

A consequence of this rule is that a name which is made directly accessible by a use clause cannot hide another name. This is quite essential for maintainability reasons: assume for example that the specification of the package D is modified to include the declaration of some new entity called X. This should normally have no effect on the procedure Q. In particular the inner reference to X should retain its meaning and should hence mean Q.X before and after the modification. (Note that we have only reduced the magnitude of this general problem since a later introduction of W within D would conflict; we believe the complete solution to be in maintenance tools.)

A similar maintainability argument leads us to reject a *unique visibility* rule, i.e. a rule forbidding redeclarations of already visible identifiers. If redefinitions of identifiers were not allowed, the later introduction of some entity named X in the declaration list of P would force textual modification of a procedure like Q, which should normally be unaffected by this change.

Note that use clauses may be viewed as a form of the import directives mentioned in section 9.3.1. However the items listed in use clauses can only be names of modules and the risk of long use lists is correspondingly reduced. Naturally, effective modularization will depend upon the user writing modules in such a way that related definitions are in the same module; related definitions will usually be required together.

9.3.3 Restricted Program Units

Whereas we subscribe to the classical rule of inheritance as a default rule, it is nevertheless important to provide means for controlling this inheritance.

We have already shown in the chapter on modules that they can be used as a powerful structuring tool for scope control. The basic ability provided by modules is the identification of a visible part, well isolated from the hidden information contained in the module body and possibly also in the private declarative part. Used in conjunction with the naming convention, modules may thus be used to discipline the use of names and to avoid *saturation* of the name space.

An additional capability is provided in order to control explicitly the inheritance of names defined in outer units. This capability is called a *visibility restriction* and the subprograms and modules which have such a restriction are said to be *restricted* program units.

The main purpose of a visibility restriction is to limit the intrusion of names pervading from outer units. This will be achieved by stating explicitly what environment should be inherited. In general a visibility restriction has the form

restricted [(unit_name {,unit_name})]

where the first name may be the name of an enclosing program unit but otherwise the names are names of modules.

The usual reasons for avoiding long name lists apply to visibility restrictions as well. For this reason the names that may appear in visibility restrictions are those of program units rather than those of individual variables. Precise control of inheritance should be obtained by grouping the elements that are likely to be imported together into a single module.

The following example containing the procedures P, Q, R and the modules D, E, F is used to illustrate the possible cases. In addition L is assumed to be the name of a library module.

```
package D is
    T, U, V : BOOLEAN;
end D;

procedure P is
    use D;
    -- local declarations of P

    package E is
      B, W, V : INTEGER;
    end E;

    procedure Q is
        use E;
        -- local declarations of Q

        package F is
          ...
        end F;

        procedure R is
          ...
        end R;
    end Q;
end P;
```

Consider now the various forms of visibility restrictions that can prefix the procedure R. The simplest, and most radical, restriction is as follows:

```
restricted procedure R is
  ...
end R;
```

In this form, R inherits no names from outer units apart from the predefined identifiers. In particular the names of the packages D, E, F are not inherited and the use clauses given for D and E in outer units are inoperative within R.

The effect of the visibility restriction is thus to restart with a completely *fresh* name space in conditions similar to those of the main program. As a next step consider

```
restricted (D, F, L)
procedure R is
  ...
end R;
```

This time the restriction includes a visibility list mentioning the names of modules, none of which is enclosing R itself. The effect of the restriction is as before to start with a fresh name space apart from the listed names D, F, L. As before, any use clause given outside R is inoperative within R. The appearance of the module names D, F, L in the visibility list means that these names can appear within R in use clauses and can serve to form selected components such as D.T.

Consider finally the case where the first name mentioned in the visibility restriction is that of a unit textually enclosing R

```
restricted (Q, D, L)
procedure R is
   ...
end R;
```

The mention of the enclosing program unit Q in the visibility list means that R does not see any name, apart from the module names D and L, beyond the names declared in Q itself. Hence the only names inherited by R are Q, F, those introduced by the local declarations of Q, and the names D and L. As before, any use clause given outside R is inoperative within R, in particular that of E.

As a final example consider the following procedure:

```
procedure P (X : INTEGER) is

    LP : BOOLEAN;

    package E is
        LE : BOOLEAN;
        ...
    end E;

    package body E is
        LBE : BOOLEAN;
        ...
    end E;

    procedure Q (Y : INTEGER) is
        LQ : BOOLEAN;
        ...

        restricted (Q, E)
        procedure R (Z : INTEGER) is
        begin
          -- The local identifiers of Q are visible:
            -- the procedure Q itself
            -- the parameter Y
            -- the local variable LQ
            -- the procedure R itself
            -- the parameter Z of R

          -- The package name E is visible:
            -- names such as E.LE can be used
            -- the clause "use E;" is legal
            -- as usual, LBE is not visible

          -- The names P, X, and LP are not visible
        end R;
    begin
        ...
    end Q;

begin
    ...
end P;
```

9.3.4 Summary of the visibility rules

In order to define the set of identifiers that are visible at a given program point we have to consider:

(a) The set E of identifiers inherited from enclosing units: The units to be considered are all units enclosing the point considered, up to the immediately enclosing restricted unit or the compilation unit itself. The set E can be built layer by layer starting from the latter unit. It is initialized with the predefined identifiers. Each layer adds to E the identifiers that it declares. If an identifier added by the layer considered is already in E, the new meaning is retained.

(b) The list M of modules found so far in use clauses:
This list is the union of the modules found in the use clauses of the enclosing units up to the immediately enclosing restricted unit or the compilation unit itself.

The set of identifiers that are visible at a given program point contains the set E and the identifiers satisfying the two following conditions:

(1) They are declared in one and only one of the modules of M

(2) They are not already in E

The latter condition means that an identifier made directly visible by a use clause cannot hide any other identifier.

9.3.5 Scope rules for enumeration and record types

Character types are handled as enumeration types and there may be several character types defined within a program. As a consequence, a character literal such as "A" may belong to several enumeration types. For uniformity, the same identifier may also appear in several enumeration types.

```
type COLOR  is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN);
type LIGHT  is (RED, AMBER, GREEN);
```

Here references to the literals RED and GREEN require a qualified expression such as LIGHT (GREEN) when the context is insufficient to determine the type.

An essential requirement of the scope is that it must not be possible to declare a variable with the name of any enumeration value. If this were allowed, the declaration

```
BROWN : COLOR;
```

would give rise to ambiguities. For instance when comparing another variable to BROWN it would not be possible to know whether the literal or the variable were meant.

Note on the other hand that redeclaration of an enumeration type is no problem. If the type COLOR is declared in a procedure P and then redeclared in an inner unit, names in the form of selected components (for example P.COLOR) can be used to refer to the outer definition.

A record type definition introduces a new scope. Hence component identifiers may be freely chosen. The record scope is opened for each selection by the dot following the name of a record variable in the selected component.

Enumeration literals within a record are not just within the scope of the record but within the scope of the innermost enclosing block, subprogram or module. Hence with

```
type PERIPHERAL is
   record
      UNIT : (PRINTER, READER, DISK, DRUM);
      ...
   end record;

READER : INTEGER;          -- illegal declaration
```

the declaration of the variable READER is invalid since it clashes with the declaration of the enumeration value READER within the record. However, the declaration UNIT : INTEGER; would be valid (but no doubt misleading).

As with Pascal, variants within a record do not introduce a new scope. Hence the component names of every variant must be distinct, even if they are semantically equivalent as far as the programmer is concerned. The reason for not introducing a new scope with a variant can be seen from the following example:

```
type T is
   record
      DISCRIMINANT : constant BOOLEAN;
      case DISCRIMINANT of
         when TRUE  => A : REAL;
         when FALSE => A : INTEGER;        -- illegal declaration of A
      end case;
   end record;

R : T;
```

A selected component such as R.A would have to be treated as a conditional expression possibly delivering results of alternative types.

9.3.6 Renaming

A renaming capability is offered in the Green language. As an example consider

```
declare
   L   : PERSON renames LEFTMOST_PERSON;
   R   : PERSON renames TO_BE_PROCESSED(NEXT);
begin
   L.AGE  := L.AGE  + 1;
   R.AGE  := R.AGE  - 1;
   if L.BIRTH < R.BIRTH then
      L.RANK  := L.RANK  + 1;
   else
      R.RANK  := R.RANK  + 1;
   end if;
end;
```

The renaming declarations of L and R are used to introduce new local names for the outer variables LEFTMOST_PERSON and TO_BE_PROCESSED(NEXT). In the sequence of statements of the block, L and R may be used as convenient names of the variables that they represent. In this sense, the renaming facility can be used for purposes similar to the Pascal *with* statement as a convenient alternative for frequently used long names. However, components of renamed records are still designated with the syntax of record components, thus avoiding possible confusion with variables bearing the same name as the components.

In addition to the notational advantage, a renaming declaration avoids re-evaluating the access path to a record variable for each component selection, and thus simplifies the generation of efficient code.

Renaming declarations are also possible for subprograms, modules, and exceptions. In addition subtype declarations can be used to rename types.

```
procedure MR renames MULTIPLEXER.READ;

task L_C renames LINK_CONTROLLER(6);

I_O_MALFUNCTION : exception renames INPUT_OUTPUT.MALFUNCTION;
```

The ability of renaming turns out to be essential when working with modules that are developed independently by different groups of programmers. Being independently developed, such modules may declare the same identifiers. If later they appear in a use clause within a given unit it may often be convenient to resolve the name clashes by renaming rather than using the dot notation whenever these identifiers appear. For example consider

```
package TRAFFIC is
   type COLOR is (RED, AMBER, GREEN);
   ...
end TRAFFIC;
```

```
package WATER_COLORS is
   type COLOR is (WHITE, RED, YELLOW, GREEN, BLUE, BROWN);
   ...
   end WATER_COLORS;

declare
   use TRAFFIC, WATER_COLORS;
   subtype T_COLOR   is TRAFFIC.COLOR;
   subtype W_COLOR   is WATER_COLORS.COLOR;
   X   : T_COLOR;
   Y   : W_COLOR;
   ...
begin
   ...
end;
```

The subtypes T_COLOR and W_COLOR rename the corresponding types and are unambiguous within the block, (whereas COLOR would be ambiguous).

Because of the possibility of overloading, it will often suffice to rename conflicting type names; names of subprograms being thereafter resolved by the overloading rules. The renaming facility can also be used to provide a name more appropriate to the context of its use. For instance, the author of a sort routine may call his version QUICKSORT2 whereas SORT may be better (and less cumbersome) throughout the application.

9.3.7 Linear Elaboration of Declarations

For reasons of readability (not for compilation reasons), the elaboration of declarations is performed linearly in a *one-pass* fashion. This means that the scope of a declaration extends from the point where the declaration is made (including the declaration itself), to the end of the unit containing the declaration. This simple rule has the advantage of corresponding to the understanding that a reader may have of the text when reading the program sequentially.

For constant declarations, the one pass strategy is quite intuitive. It permits later type and constant declarations to be expressed in terms of a previous constant declaration, as in the following example where each declaration is made in terms of the one on the previous line.

```
declare
   LENGTH   : constant INTEGER := 100;
   SQUARE   : constant INTEGER := LENGTH*LENGTH;
   subtype SURFACE is INTEGER range 0 .. SQUARE;
   S,T,U    : SURFACE;
begin
   ...
end;
```

The one-pass strategy has the other advantage of avoiding cyclic constant declarations. However *a precaution is needed for inner scopes*

```
declare
   N     : constant INTEGER := 10;
begin
   ...
   declare
      M  : constant INTEGER := N**2;        -- value equal to 100
      N  : constant INTEGER := 20;          -- illegal redeclaration!
   begin
      ...
   end;
end;
```

Whereas redeclaration of an identifier is generally permitted, in the above case the redeclaration of N is illegal and shall be rejected by the translator, since the value of the outer N has already been used in the same declarative part. The rule applied is as follows:

> *Identifiers of an outer scope which are used in a given sequence of declarations may not be redefined in that sequence of declarations.*

Linear elaboration means that mutually recursive procedures must have their specifications appearing before their bodies. Similarly, this means that the module bodies of two packages (or tasks) can mutually refer to their visible parts, provided that their module specifications appear before, as in the following example:

```
package A is
   ...
end A;

package B is
   use A;
   ...
end B;


package body A is
   use B;
   ...
end A;

package body B is
   use A;
   ...
end B;
```

If two access types are mutually dependent, an incomplete declaration of one of them must be given before the corresponding type declaration appears in full. This is illustrated in the following example:

```
type CAR;        -- incomplete declaration of CAR

type PERSON is access
   record
      NAME    : STRING(1 .. 20);
      AGE     : INTEGER range 0 .. 130;
      SPOUSE  : PERSON;
      VEHICLE : CAR;
   end record;

type CAR is access
   record
      NUMBER  : INTEGER;
      OWNER   : PERSON;
   end record;
```

If the incomplete predeclaration of CAR were omitted, then in the presence of an outer identifier CAR of any type, the wrong association would be made when encountering the declaration of the component VEHICLE, but this would be trapped when the definition of the access type CAR was encountered. On the other hand, if an outer type CAR existed (and the inner one did not) it is clear that the association with the outer one would be valid.

The importance of the predeclaration can be seen by noting the assumptions that the reader who has read the definition of PERSON but not that of the inner definition of CAR would make in the absence of the predeclaration.

10. Separate Compilation and Libraries

10.1 Introduction

Separate compilation of program units is a practical necessity. Its basic goal is to permit the separation of large programs into simpler, more manageable parts and to provide a library facility. Separate compilation helps to reduce compilation costs and to simplify development and management of program corrections and updates.

For large projects involving several programmers, separate compilation enables a physical separation of program texts in a way that reflects the division of work and responsibilities. Once the common interface between two parts has been agreed upon and recorded the two parts can be developed and compiled separately. The fact that the common interface is a physically separate text guarantees that separate recompilation of either part does not invalidate the common interface.

The physical separation of program texts may be viewed as a support facility for the structured programming concept of refinement. It may also be used to conceal the text of subprograms from users who are only allowed to call them. Such concealment may be justified either by confidentiality or in order to prevent inference of implicit assumptions regarding the functioning of the subprograms. Finally this physical separation enables the construction of libraries.

It is appropriate at this stage to introduce the distinction between *independent* and *separate* compilation, a distinction first made by J.J. Horning.

Independent compilation has been achieved by most assembly languages and also by languages such as Fortran and PL/1. Compilation of individual modules is performed independently in the sense that such modules have no way of sharing knowledge of properties defined in other modules.

Independent compilation is usually achieved with a lower level of checks between units than is possible within a single compilation unit. In consequence, independent compilation came into disrepute and was rejected by safety minded, early typed language definitions such as Algol 68 and Pascal. Fast compilation of the complete program was often advocated by promoters of these languages as a safe alternative to independent compilation. Fast compilation, however, has its limits, and it fails to answer the needs of confidentiality and libraries.

Separate compilation, on the other hand, reconciles type safety and the pragmatic reasons for compiling in parts. It is based on the use of a *library file* which contains a record of previous compilations of the units which form a program. It has been developed in the language Sue [CH 72] and in later languages such as Jossle [PW 74], Lis [IRHC 74], Mesa [GM 77, LS 79] and recent extensions of Algol 68. We next discuss its properties in terms of what is proposed in the Green language.

When a program unit is submitted to the translator, the translator also has access to the library file. Consequently, it is able to perform the same level of checking, in particular type checking, whether a program is compiled in many parts or as a whole. It is indeed the existence of the library file that makes the compilation separate but not independent.

Using the general information available in the library file, the translator will be able to help the user manage recompilations. In particular it will be able to display information about the current compilation state of a program divided in several compilation units: which separate program units have been compiled, and which need to be recompiled because of prior recompilations.

It is thus for reasons of safety and practicability that the Green language has a strong facility for separate compilation. Two additional criteria have been followed in this design, namely simplicity of use and simplicity of implementation.

Separate compilation being a user oriented facility, it should be very simple to understand and to use. Consequently it should not introduce other concepts than those required by the nature of separate compilation. Scope rules and the general form of separately compiled program units should be the same as those of normal program units.

The design must also be compatible with having a simple translator. The additional work required for separate compilation should stay within reasonable limits, since one of the goals is to save globally on compilation and recompilation time.

10.2 Presentation of the Separate Compilation Facility

A complete program is either a procedure body or it is a succession of compilation units submitted individually or together to the compiler. In the Green language the major forms of program units, that is subprograms and modules, can be compilation units. In addition, a module body can be compiled separately from the corresponding module specification.

For program units that are separately compiled it is especially important to exercise precise control over the names that are visible. Consequently compilation units are restricted program units, and the visibility rules that apply to them are those that apply to all restricted program units.

Although compilation units are submitted individually to the translator they can depend on each other through their visibility lists. For this reason the compilation units which form a given program are said to belong to a common *program library*.

Traditionally, one can distinguish two main styles of program development, top-down (or hierarchical) program development and bottom-up program development. The separate compilation facility should support both styles, and also any intermediate form.

10.2.1 Bottom-up Program Development

For this form of program development we may have programmers developing libraries of generally usable modules. The module construct, especially in the package form, is the main construct to be used for this style of program development.

Each generally usable package can (and should) be compiled separately and made thus available in the program library. The specification and the body (if any) of such a package can be compilation units and they can be submitted either in the same or in two different compilations.

Some packages produced in this form of program development do not depend on any outside information, except perhaps that of the predefined environment; for example, a package of physical constants. More generally, packages may depend on information defined by other modules of the program library. For example an application level input output package may depend on a more basic input output package; similarly a surveying package could depend on this application level input output package and on another package defining trigonometric functions.

Compilation units are interdependent through their visibility lists. This is shown in the example below:

```
restricted(MATH_LIB, TEXT_IO)
procedure QUADRATIC_EQUATION is
   use TEXT_IO;
   A, B, C, D : FLOAT;
begin
   GET(A); GET(B); GET(C);
   D := B**2 - 4.0*A*C;
   if D < 0.0 then
      PUT ("IMAGINARY ROOTS");
   else
      declare
         use MATH_LIB;  -- note: SQRT is defined in MATH_LIB
      begin
         PUT ("REAL ROOTS : ");
         PUT ((B -  SQRT(D))/(2.0*A));
         PUT ((B +  SQRT(D))/(2.0*A));
         PUT (NEWLINE);
      end;
   end if;
end QUADRATIC_EQUATION;
```

For the programmer writing QUADRATIC_EQUATION, this procedure represents his *complete* program. However, in order for that program to work, the program library should already contain the two packages MATH_LIB and TEXT_IO on which QUADRATIC_EQUATION depends. Otherwise the function SQRT supplied by MATH_LIB and the procedures GET and PUT supplied by TEXT_IO will not be visible. The procedure could only be considered as complete and not dependent on any other module if the packages that it used were local packages, and if in consequence it had no visibility list.

Realizing that this program might be generally usable, the programmer may decide to encapsulate it within a package, perhaps along with other similar procedures.

```
package EQUATION_SOLVER is
   procedure QUADRATIC_EQUATION;
   procedure LINEAR_EQUATION;
end;
```

```
restricted(MATH_LIB, TEXT_IO)
package body EQUATION_SOLVER is

   procedure QUADRATIC_EQUATION is
      -- same text as before
   end;

   procedure LINEAR_EQUATION is
      -- reads a linear equation, solves it, prints results
   end;
end EQUATION_SOLVER;
```

A program using this is shown below:

```
restricted(EQUATION_SOLVER)
procedure MAIN is
   use EQUATION_SOLVER;
begin
   for I in 1 .. 10 loop
      QUADRATIC_EQUATION;
   end loop;
end MAIN;
```

Note that the program MAIN need only mention the package EQUATION_SOLVER in its visibility list. It need not (and should not) mention the packages MATH_LIB and TEXT_IO, which are actually needed by the package body of EQUATION_SOLVER, since MAIN does not contain direct calls to subprograms defined in either MATH_LIB or TEXT_IO.

The outermost declarations made in compilation units introduced in this fashion must be elaborated before execution of the program. When preparing an object program for execution the translator (and/or linkage editor) must collect all modules called directly or indirectly and elaborate them before program execution. In the example above of the procedure MAIN, the modules EQUATION_SOLVER, MATH_LIB, TEXT_IO (and possibly also other modules used by the latter) must be elaborated. Elaboration of these modules must proceed in an order consistent with the partial ordering discussed in 10.2.3.

Note finally that separately compiled modules may also be generic. Instances of such generic compilation units can be obtained as usual:

```
restricted(INPUT_OUTPUT)
procedure TREAT_ELEMENTS is
   type ELEMENT is ...
   package ELEMENT_IO is new INPUT_OUTPUT(ELEMENT);
   -- use of the input output procedures for objects of type ELEMENT
end TREAT_ELEMENTS;
```

10.2.2 Hierarchical Program Development

The other form of program development is called hierarchical or top down, as used in programming by stepwise refinement [Wi 71, Wo 72]. The top level provides a formulation of the program in terms of operations to be supplied by the next lower level. Each such operation is then further defined in terms of operations of another lower level, etc. Corresponding to this form of program development the Green language offers the possibility of having compilation units that are *subunits* of other compilation units.

We illustrate subunits by means of a variant of the example of section 10.2 of the Reference Manual. Assume that we are developing the procedure TOP in a top down fashion. The top level definition is given by the following compilation unit:

```
procedure TOP is

   type REAL is digits 10;
   R, S : REAL;

   procedure Q(U : REAL);

   package D is
      PI : constant REAL := 3.14159_26536;
      function    F(X : REAL) return REAL;
      procedure   G(Y, Z : REAL);
   end D;

   package body D is separate;              -- stub of D
   procedure Q(U : REAL) is separate;       -- stub of Q

begin  -- TOP
   ...
   Q(R);
   ...
   D.G(R, S);
   ...
end TOP;
```

The specifications of the procedure Q and of the package D are given as usual. Hence the statements of TOP can be defined in terms of these units, that is, the procedure Q and the subprograms F and G defined by the package D can be called. In this case however the bodies of Q and D have not been provided, but only body stubs in their stead, so that these two program units can be compiled separately as subunits of the procedure TOP. The procedure Q appears as

```
restricted (TOP)
separate procedure Q(U : REAL) is
   use D;
begin
   ...
   U := F(U);
   ...
end Q;
```

Although separately compiled, Q must still have access to the identifiers declared within TOP. For example it must see the type REAL and the package name D. This is achieved by the visibility list **restricted** (TOP) mentioning the name of the *enclosing* procedure TOP. Similar considerations apply to the separately compiled body of the package D:

```
restricted (TOP)
separate package body D is
   -- some local declarations of D followed by

   function F(X : REAL) return REAL is
   begin
      -- sequence of statements of F
   end F;

   procedure G(Y, Z : REAL) is separate;   -- stub for G
end D;
```

In this case the package body contains the body of the function F and, again, a stub for the procedure G which is thus a subunit of D:

```
restricted (TOP, INPUT_OUTPUT)
separate procedure G(Y, Z : REAL) is
begin
   -- sequence of statements of G
end G;
```

Subunits can be declared at the outermost level of another unit or subunit. This creates the possibility of a hierarchy of program subunits depending on a given compilation unit. This hierarchy is no different from the nesting hierarchy in ordinary program units. In particular, the visibility rules are the same and a subunit can depend on dynamic information. For example, consider

```
restricted (TOP)
separate procedure Q(U : REAL) is
   use D;
   L : constant REAL := U**2;
   procedure S is separate;
begin
   ...
end Q;
```

Access to the local constant L is still possible within S, exactly as if the body of S were textually nested at the place of the stub.

```
restricted (TOP)
separate procedure S is
begin
   -- access to L is possible
end S;
```

It should be clear that these two methods for introducing compilation units are not mutually exclusive and can be used in combination. For example, a general purpose package may be split into subunits in order to facilitate its development, compilation and subsequent recompilation.

10.2.3 Compilation Order

Compilation units may be compiled separately, but this does not mean that compilations can be submitted in an arbitrary order, since units are not independent. In particular, we have seen that one unit may appear in the visibility list of another unit, and some units can be subunits of other units. These two forms of dependence determine a partial ordering of compilations:

- A unit cannot be compiled before any unit that it sees.
- A module body cannot be compiled before the corresponding module specification.
- A subunit cannot be compiled before the unit containing its declaration (and stub).

These rules are certainly rules of common sense and the translator must enforce them. It should be clear that they only define a partial ordering and that several sequences are actually possible. For example

(a) The procedure QUADRATIC_EQUATION cannot be compiled before the specifications of MATH_LIB and TEXT_IO. On the other hand, the latter can be submitted in any order. Furthermore the package bodies of MATH_LIB and TEXT_IO need not themselves be compiled before QUADRATIC_EQUATION. Of course, in order to execute the program, it will be necessary that all compilations be completed.

(b) The procedure TOP must be compiled before its subunits Q and D can be compiled. On the other hand, the order of compilation of the procedure body of Q and of the package body of D is irrelevant. Note that the body of Q contains the clause **use D**; but this has no influence on order of compilation since all the information that Q needs to know (and can know) about D is contained in the specification of D compiled with TOP.

(c) The subunit G cannot be compiled before either the package body of D (containing its stub) or INPUT_OUTPUT (appearing in its visibility list). On the other hand the body of D and the INPUT_OUPUT package can be compiled in any order.

10.2.4 Recompilation Order

Similar considerations apply for recompilations. If a given declaration is modified in a program unit, all other units actually using the declared entity are affected by the change and should be recompiled.

In principle, a translator including a librarian facility could compare the old text and the new text and keep track of the changes on an individual basis. It would thus be able to keep recompilations to an absolute minimum.

For simpler translators however, one may assume that the smallest *grain of change* recognized by the translator is the recompilation of a compilation unit. For translators using such a strategy the rules defining the need for recompilations are identical to the rules defining the compilation order.

Given the ability to separately compile a module specification and a module body, this simple strategy should not in practice require many more recompilations than strictly necessary.

Note, in this respect, that the language design has carefully avoided *unnecessary textual dependency*. For example, the fact that visibility restrictions are given with the procedure body and not with the procedure declaration (and/or stub) is quite important. Consider the alternative

```
procedure EXAMPLE is
   package A is
      ...
   end A;
   package B is
      ...
   end B;

   -- The following is not in Green:

   restricted (A, INPUT_OUTPUT)
   procedure P(X : INTEGER) is separate;

   restricted (EXAMPLE, MATH_LIB)
   procedure Q(Y : REAL) is separate;
   ...
end EXAMPLE;
```

Assume that in some later revision of this program, input output needs to be performed within the body of Q. Then if visibility lists were provided with the stubs, it would be necessary to modify the stub of Q and hence to recompile the text of the EXAMPLE. However since the stub of P is also provided there (this is the textual dependence), a translator using the simple strategy would conclude the necessity of recompiling P as well.

While we recognize that future translators might adopt more ambitious schemes, this design has carefully avoided any feature that would not be compatible with the simple strategy. Given this careful avoidance of unnecessary textual dependencies the number of recompilations can be kept quite close to the actual minimum.

10.2.5 Methodological Impact of Separate Compilation

The ability to separately compile a module specification and the corresponding module body has important methodological consequences for program development and maintenance. For example, it allows a team of programmers to agree upon a common interface and define it by one or more package specifications. This being done, the package bodies and any other unit using the common interface can be developed in parallel and compiled in an arbitrary order. Consider for example the specification of table management package given in section 7.5 of the Reference Manual:

```
package TABLE_MANAGER is
  type ITEM is
    record
      ORDER_NUM   : INTEGER;
      ITEM_CODE   : INTEGER;
      ITEM_TYPE   : CHARACTER;
      QUANTITY    : INTEGER;
    end record;

  NULL_ITEM : constant ITEM :=
    (ORDER_NUM | ITEM_CODE | QUANTITY => 0, ITEM_TYPE => ' ');

  procedure INSERT   (NEW_ITEM   : in  ITEM);
  procedure RETRIEVE (FIRST_ITEM : out ITEM);
  TABLE_FULL : exception;  -- may be raised by INSERT
end TABLE_MANAGER;
```

The above package specification contains all the declarations that need to be known by any unit using the services of the table manager.

It is important to realize that the package body of TABLE_MANAGER may be modified and recompiled without need for recompilations of units using TABLE_MANAGER. As long as the operations promised by the visible part of the definition module are correctly achieved, the user will not be affected by such changes. Another version of the package body using a different technique (such as a balanced tree instead of lists) may be substituted without affecting the user.

This ability to compile a module specification and the corresponding module body separately is a logical extension of the idea of encapsulation. Since the user is not affected by the contents of the package body (provided it is correct), there is no need to show him its text: all the user needs is an object module.

Separate compilation of module bodies may thus be used to achieve *physical hiding*. This will be useful for confidentiality purposes. It will also help to deny the user the possibility of reading the algorithms and inferring implicit assumptions that might not be satisfied by later implementations. In this sense separately compiled module bodies provide good support for the policy of restricted flow of information advocated by Parnas [PA 73]. They can be used in conjunction with methodologies such as algebraic specification [GU 77].

10.3 The Program Library

In the previous sections we have shown how a large program can be separated into several compilation units. All such units logically belong to what is called a program library. Such library may contain the units necessary for a single *main* program, but it may also contain the units of several main programs, especially in the case of related projects. This should be left as a user decision.

Associated with each program library, there must be a *library file* that records information relative to the compilations that have already been done. In particular, the library file must contain *symbol tables* for separately compiled module specifications. It must also record compilation dates, and the unit-subunit relations, in order to enable the translator to check the order of compilation.

When submitting a compilation unit to the compiler, the programmer must provide

- The source text of the compilation unit
- The name of the library file to which the unit belongs

It is this second item that makes the compilation separate but not independent; the library file enables the translator to perform type checking across separately compiled units exactly as within a given compilation unit.

The concept of program library as defined above is not much different from the usual concept of library, provided that means exist for transferring units from one library to another one. Such facilities are not properly within the domain of the language but rather within that of its support environment. We can only state here desired facilities that this environment should provide. When standardized, these facilities could be specified by appropriate pragmas:

Library creation:

> There should be a command for library creation. Some translators may decide to initialize the library with the predefined modules upon library creation.

Inclusion of library units:

> There should be a command to include a unit of one library into another library. Note that such inclusion requires inclusion of the information pertaining to the unit (e.g. its symbol table). After its inclusion, a unit should be indistinguishable from other units of the library. Inclusion of a unit may require inclusion of units that it sees.

Deletion of library units

> Conversely there should be a command to delete a unit from a given library.

Completion check

> There should be a command by which a programmer states that the program is complete (all units have been compiled). The translator will check that this is the case and issue appropriate error messages otherwise. Similarly this command (or a distinct one) could be used to display global information about the current state of the program library: which units have been compiled, which subunits have never been compiled, which units need to be recompiled, etc.

Completion and status checks are quite useful since a library may contain obsolete units at intermediate stages of the program development.

Since the translator is able to detect the need for recompilations, it could conceivably do these automatically when needed. However changes are often done for several units at the same time. A translator that performed recompilations after each change might perform more recompilations than necessary unless it had global knowledge of all changes submitted.

Assume for example that the specifications of the packages A and B are modified. If all units seeing A were automatically recompiled, then if some of them also see B, they would be recompiled a second time after the compilation of B.

Hence it is certainly preferable to let the user manage the recompilations. However, this means that tools for displaying the current status of compilation units of a program should be provided. Similarly it means that the user should be able to state that a program is complete and let the translator check that this is actually the case.

10.4 The Implementation of Separate Compilation

The purpose of this section is to demonstrate that the proposed language features can readily be implemented safely and at a reasonable cost for the simple strategy where the minimum grain of change recognized by the translator is a unit compilation or recompilation. The model described below should not be considered as the only possible implementation technique but rather as a feasibility proof. It is similar to the technique used for Lis compilers.

As mentioned before, separate compilation of a program split into separate units involves a library file recording information on the units and on relations between them. In a sense, the library file plays a role similar to that of the symbol table for a single compilation unit. The basic idea of maintaining a library file is not widely known, but has been in use for many years for the compool facility in Jovial [PE 66].

10.4.1 Principle of Separate Compilation

A library file is essentially a representation of the declarations encountered in the outer declarative part of the compilation unit considered. These declarations may be accessed by other compilation units. In particular access may be made to:

- Any unit (or subunit) containing declarations of local subunits.
- Any separately compiled module specification since it may appear in the visibility lists of other compilation units.

As a consequence the corresponding symbol tables must be kept in the library file. It is the translator's responsibility to maintain these symbol tables. In order to perform a given compilation, it must first select the symbol tables corresponding to the current scope and then load them. In other words, it must construct a symbol table equivalent to the symbol table at a given program point as if the program was compiled as a complete text.

In order to perform this task it is useful to consider the following forest structure, which reflects declaration of units and subunits:

(a) A unit is a root.

(b) The sons of a given compilation unit are the subunits whose stubs are given within the unit considered.

This structure is necessary for the determination of scope rules. Hence it must be recorded in the library file and updated as new stubs of subunits are encountered, or as new units are compiled.

Finally for all units and subunits, a list of the modules mentioned in their visibility lists must be recorded.

The forest structure will help for the determination of the symbol tables to be loaded, for checking the validity of visibility restrictions and for the determination of the recompilations that need to be done as a consequence of previous recompilations. Naturally, the translator may use this information to assist the user with recompilations.

To check for required recompilations, the translator may use a system of time-stamping that records the order of compilations: a compilation date is associated with the symbol table of each unit or subunit.

10.4.2 Details of the Actions Performed by the Translator

The following major actions must be performed during the translation of a compilation unit:

Determination of the compilation context

During a preliminary scan of the text of the compilation unit (for instance, during the lexical or context-free analysis phase), the name of the compilation unit is recognized and a merged list of all visibility lists found in the inner declarative parts of the unit is constructed. Note that the recognition of the compilation unit name will sometimes be only possible by using the contents of the visibility lists.

Using the forest structure, the genealogy of the compilation unit may be found: its father, grandfather, and so on, until a unit (root) is found. This genealogy together with the merged visibility restrictions comprise the compilation context.

Checking the validity of the compilation context

Any unit mentioned in a visibility list within a given compilation unit must be either a library module or the root of the genealogy. Any subunit procedure mentioned in a visibility list must belong to the genealogy of the current compilation unit. Any subunit module mentioned in a visibility list must have been declared in one of the units or subunits of the genealogy.

The remaining names may be those of procedures and modules that are local to the current unit. This sublist is kept for later checks.

The following checks of compilation dates must be performed:

- In the genealogy, each son must have been compiled after its father.
- Each compilation unit of the genealogy must have been compiled after the modules it mentions in its visibility list.
- Each module of the merged visibility list must have been compiled after the modules it mentions on its own visibility list.

Translation may only proceed if all these checks are successful. Otherwise diagnostics, a list of required recompilations and a recommended order must be printed by the translator.

Table loading

The symbol tables of the portion of the genealogy used and those of the modules of the merged visibility lists may now be assembled (of course, actual accessibility to the declarations contained in a module must be given only in scopes where the module name appears in a use clause or in selected components).

This table assembly may involve establishing some links between the different tables since they may refer to one another (for instance an identifier declared in a given package may be of a type declared in another package).

Update of the forest structure, table unloading

At the end of the translation of a unit or subunit, the date of compilation must be updated. For a module, or for a procedure containing local declarations of separate units, a new symbol table must be stored in the library file in a suitable format. Newly declared separate units must be entered in one of the trees of the forest. If a new unit is compiled a root must be added to the forest.

When a module is recompiled it is possible to use the forest structure to mark all units using the module as needing recompilation. Similarly when a procedure or module is recompiled, all subunits declared within it may be marked as needing recompilation.

10.4.3 Treatment of Module Bodies

For a given module, the two disjoint units (specification and body) must be viewed as defining (complementary aspects of) the same logical entity. Consequently it will be convenient for the user to have a single object module, and not two. In order to achieve this effect the code produced during the compilation of the module specification, if any, may be kept in some intermediate form in the library file entry associated with the module. Later, when compiling the module body, this initial code may be recovered and the compilations may proceed as if the two units were concatenated.

10.4.4 Summary of the Information Contained in a Library File

The library file contains a representation of the forest structure discussed above. Each node of a tree corresponds to a subunit, except the root, which is always a unit. A node contains:

- The name of the unit or subunit.
- Its nature: subprogram, package or task.
- Its compilation date and that of the associated module body, if there is one.
- The list of modules mentioned in the visibility list.
- A symbol table, if the unit or subunit contains declarations of subunits, or if the unit or subunit is a module specification.
- A boolean component indicating need for recompilation.

The entry for a given node is created either when the stub for a subunit is analyzed (and then initialized in the state "recompilation needed"), or, when the node is for a unit, during its compilation. This entry is updated during compilations. The entry for a subunit may be deleted from the compilation file if its declaration is absent from the parent unit (or subunit), when recompiling the latter.

Each individual symbol table must be kept in a format that simplifies the reestablishment of the relations between different symbol tables when they are assembled. As an example, consider the two following packages:

```
package D is
   type T is ...
   ...
end D;

restricted(D)
package E is
   use D;
   X : T;
   ...
end E;
```

Given the symbol table entry for the declaration of X, it must be possible to find the symbol table entry for its type T.

If internal references are used to represent such relations, they must be relocated when the symbol tables are assembled. Solutions involving relocation information, or alternatively general tables of types, have been used in the past by implementations of languages supporting separate compilation.

Note, finally, that symbol tables may be transferred from the library file of one program to that of another program. The internal structure adopted for symbol tables should permit this.

10.5 Summary and Conclusion

To summarize the Green separate compilation facility:

- Compilation units can be subprograms, module specifications and module bodies. All compilation units are restricted program units. The compilation units of a program form a program library.
- Subunits of other compilation units can be defined by using body stubs. These subunits are separately compiled.
- The visibility rules applicable to compilation units and subunits are the usual scope rules applicable to all restricted program units. The order of compilation and recompilation is only governed by these rules.

We have attempted to restrict the number of language concepts to the minimum required by the nature of separate compilation, and believe we have produced quite a simple solution. Naturally this simplicity was obtained by treating separate compilation as one of the major goals in the language design, and not as an afterthought.

Separate compilation has been conceived mainly as a user facility supporting the traditional forms of program development.

This language proposal can be implemented at very reasonable cost, as evidenced by the previous sections and by previous languages supporting similar separate compilation facilities. In consequence the type rules may be enforced across separate units to the same degree as within a given unit.

Separate compilation in no way changes the meaning of a program. Furthermore we have demonstrated that the information contained in a library file may be used to check that a given compilation does not use information from other units that have in the meantime become obsolete.

Finally, one of the motivations of separate compilation is the creation of software libraries. This is supported by the present proposal. By far the most useful library units should be packages. The proposed facility permits their use with the same degree of safety as for internal units.

It is expected that library packages will be used for encapsulation of type definitions, for common constants and data, and for shared declarations. The fact that these library items are already compiled program units and not source texts offers a degree of safety not found in languages providing merely independent compilations.

Other modules will be used for the creation of user packages such as input output packages, to be found in libraries. The ability to compile a module specification separately from the corresponding module body provides the possibility of separating the interface of a module from its implementation. Thus it supports information hiding and reliability to an extremely high degree.

11. Tasking

11.1 Introduction

Tasking is an important aspect of many embedded systems and this importance is clearly recognized in the Steelman requirements. However it seems to have been neglected in most languages currently in production use for such systems. One reason has clearly been a lack of confidence in the many different facilities put forward for the control of parallelism. Semaphores, events, signals and other similar mechanisms are clearly at too low a level. Monitors, on the other hand, are not always easy to understand and, with their associated signals, perhaps seem to offer an unfortunate mix of high level and low level concepts. It is believed that Green strikes a good balance by providing facilities which are not only easy to use directly but can also be used as tools for the creation of mechanisms of different kinds.

The basic textual concept in Green is that of a task which in form is closely analogous to a package module. Statements enable tasks to be initiated in parallel with their parent task and with each other. The termination of tasks basically follows the scope structure but mechanisms are also provided to allow wayward tasks to be controlled.

Communication and synchronization are both achieved using the concept of a rendezvous between a task issuing an entry call and a task accepting the call by an accept statement. An entry call is similar to a procedure call except that the calling and called tasks are distinct and synchronized.

Great power is provided by the select statement which enables a task to respond to several different possible entry calls.

Other facilities include a delay statement, which, combined with the select statement, provides a time-out mechanism in a natural manner. Interrupts may be handled by a representation specification associated with a particular entry.

This chapter starts by describing the facilities and illustrating their use with examples. This is followed by a brief historical survey of parallel processing mechanisms which put the present Green operations into perspective. A substantial example is then given. A further section covers miscellaneous points of rational that are not covered elsewhere, and a final section discusses some implementation considerations.

11.2 Presentation of the Tasking Facility

This section introduces the tasking facilities, defines them generally, and illustrates them by means of examples. After a presentation of tasks and their associated hierarchy, we describe rendezvous, entry calls, and accept statements. The discussion continues with select statements, delay statements, and interrupts, whose occurrences can be viewed as implicit entry calls. Families of tasks and entries, and generic tasks are then described and the presentation concludes with the use of these concepts for scheduling.

11.2.1 Tasks : Textual Layout

A task is a textually distinct program unit which may be executed concurrently with other tasks. It is very similar in form to a package module. Indeed the major difference between a package module and a task module is that the former is merely a passive construct whereas a task may be active.

Like the package module, a task may be declared within any declarative part (except the visible part of another task) and similarly comprises two distinct pieces of text. These are the specification part which describes its external appearance, and the module body which describes its internal behavior. These two parts will often be juxtaposed in the text but need not and indeed need not be compiled together. We will now consider the details of these two parts and, in particular, show how they differ from the corresponding parts of package modules.

The specification part comprises a header giving the name of the task (and possibly other characteristics, to be discussed later, such as whether it is a family or generic) and a declarative part which describes its appearance to the outside world. This declarative part is usually known as the visible part. The visible part may contain type, subtype, constant, entry, subprogram, exception, and renaming declarations. The declarations of variables and modules are disallowed on methodological grounds (since such variables would not be controlled by the task) and because of the difficulty of preventing access to them if the task is not active. Entries externally look like procedures but are executed in mutual exclusion.

It is the possibility of the inclusion of entry declarations which distinguishes the visible part of a task from that of a package module.

The following is an example of the specification part of a task:

```
task LINE_TO_CHAR is
   type LINE is array (1 .. 80) of CHARACTER;
   entry PUT_LINE  (L  : in  LINE);
   entry GET_CHAR (C  : out CHARACTER);
end;
```

The module body of a task has a form similar to that of a package body and comprises a declarative part and a sequence of statements. The body of the above example has the following outline.

```
task body LINE_TO_CHAR is
   BUFFER : LINE;
begin
   -- sequence of statements
end;
```

The full details of the body of this example are deferred until section 11.2.4.

11.2.2 Task Hierarchy

Before describing the detailed statements associated with tasks, it is important that the reader understand the underlying concept of a task, its activation, and its parent. We describe this in terms of a model which should be considered to be only illustrative.

We distinguish between a thread of control and the text of a task. A thread of control corresponds to a task activation whereas the text is merely a passive description of some code. The main program can be considered to be an anonymous task with a thread of control created by the underlying system.

When a thread of control enters a scope containing task declarations, the elaboration of each declaration creates just one new potential thread of control (or in the case of a family, one for each member of the family). There is therefore a one to one correspondence between threads of control and the elaboration of task declarations and we can loosely talk about a thread of control by the name of the corresponding task declarations.

Although each elaboration of a task declaration only gives rise to one thread (that is, a task cannot be multiply active) nevertheless there may be several threads corresponding to different coexisting elaborations of the declaration. This would occur for example if a task were declared in a recursive procedure or perhaps more likely in a procedure called reentrantly by several other tasks. The normal scope rules prevent any ambiguity because only one instance of the task elaboration is visible at any one point.

It should also be realized that a subprogram does not in any sense belong to any particular task. Whether or not it can be used reentrantly will depend upon its visibility. If it is within the body of a task which has no subtasks, then it can only be called by the embracing task. On the other hand, if it is declared in the same declarative part as several tasks, then it can obviously be called by all these tasks.

The *parent* of a task is the task whose thread of control elaborates its declaration. We recall from Chapter 7 of the Reference Manual that on entry into a scope containing a package module the visible part is first elaborated and the package body is subsequently executed in order to initialize the package. On the other hand, upon entry into a scope containing the declaration of a task, again the visible part is immediately elaborated but the task body is not executed until the task is made active by an initiate statement.

The initiate statement contains a name list indicating the tasks to be initiated. The tasks are then made active and their bodies are executed in parallel with each other and their parents. It should be carefully observed that the task performing the initiate statement need not be the parent although it often is (since the parent is the one who elaborates the task declaration, for an example see section 11.4.5).

As far as termination is concerned, a task will terminate on reaching its final **end**. When the parent task reaches the end of a scope containing the declaration of local tasks, the parent may have to wait until all the local tasks have terminated if they have not already done so.

For example consider the following task body T containing the tasks T1 and T2.

```
task body T is

   -- declarations

   task T1 is
      -- visible part of T1
   end;

   task body T1 is
      -- body of T1
   end;

   task T2 is
      -- visible part of T2
   end;

   task body T2 is
      -- body of T2
   end;
begin
   ...
   initiate T1, T2;
   ...
end;
```

Execution of the initiate statement causes T1 and T2 to be executed in parallel with T. The task T continues also and may have to wait at its final end for T1 and T2 to terminate.

Several initiate statements may occur, thus we could have written

```
initiate T1;
initiate T2;
```

However, the semantics of the two separate initiate statements is slightly different. In the case of

```
initiate T1, T2;
```

we are assured that both T1 and T2 are initiated together and therefore there is no possibility that one could call an entry in the other and find it not yet active. Of course, the initiating task T is also assured that the initiated tasks are also active before it obeys its next statement.

Initiation of a task which is already executing causes the INITIATE_ERROR exception to be raised. However a task may be initiated again once it has terminated and this will give rise to a new execution of the task.

Facilities are provided to enable a task to exert fine control over the termination of another task. The attribute T'ACTIVE is TRUE if the task T has been initiated and has not yet terminated. The special exception FAILURE may be raised in another task and this may be used to program *last wishes* followed by self termination. In desperation the abort statement may be used. Thus

```
abort T1, T2;
```

will cause tasks T1 and T2 plus any descendant tasks to be terminated unconditionally. This may as a consequence raise TASKING_ERROR exceptions in other tasks which were communicating with T1, T2 or their descendants at the time. The abort statement should not be used without due care.

The above discussion was presented in terms of several tasks executing in parallel. Whether this physically occurs depends upon the hardware. In a multiprocessor system actual parallel execution may occur whereas in a single processor system only one task can really be active. In any case a scheduler is required in order to allocate the *ready* tasks to the one or more processors. The scheduling algorithm takes tasks on a first in, first out basis within priorities. On initiation, tasks take the priority of their initiator but they can change their own priority by a call of the procedure SET_PRIORITY. The priority of a task T is given by the attribute T'PRIORITY. Priorities are provided as a tool for indicating relative degrees of urgency and on no account should their manipulation be used as a technique for attempting to obtain mutual exclusion.

Families of tasks and generic tasks are discussed in 11.2.8 and 11.2.9.

11.2.3 Visibility Rules

The usual visibility rules are applicable to tasks. As a consequence several tasks may share global variables and it is the programmer's responsibility to ensure their integrity. Of course the primary means of communication between tasks is not through the sharing of global variables (which should be done with caution) but by the use of the entry as described in the next section. However to disallow shared variables seems to be a constraint which would be unwise in some critical circumstances.

Shared variables are not marked in their declaration since a local variable of a generic module could be shared or not depending on the place where an instantiation if performed. Moreover any global variable is potentially shared and its appearance in a global declaration should itself be a sufficient warning. If it were used by a single task it could always be made local to that task.

11.2.4 Entries and the Accept Statement

As we have seen the visible part of a task may contain the specification of entries. Externally an entry looks like a procedure, takes parameters and is called in the same way. The difference lies in the internal behavior. In the case of a procedure the calling task executes the procedure body itself and the procedure body can be executed immediately. In the case of an entry the corresponding actions are executed by the task owning the entry, not by the calling task. Moreover these actions are only executed when the called task is prepared to execute a corresponding accept statement. In fact the calling and called tasks may be thought to meet together in a *rendezvous*. We will illustrate this by completing the example introduced earlier.

```
task LINE_TO_CHAR is
   type LINE is array (1 .. 80) of CHARACTER;
   entry PUT_LINE  (L  : in  LINE);
   entry GET_CHAR (C  : out CHARACTER);
end;

task body LINE_TO_CHAR is
   BUFFER : LINE;
begin
   loop
      accept PUT_LINE(L : in LINE) do
         BUFFER := L;
      end PUT_LINE;
      for I in 1 .. 80 loop
         accept GET_CHAR(C : out CHARACTER) do
            C := BUFFER(I);
         end GET_CHAR;
      end loop;
   end loop;
end;
```

The accept statement has the partial appearance of a procedure body. It can be thought of as a body to be executed where it stands in much the same way as a block can be thought of as an inline procedure without parameters.

The accept statement repeats the formal part of the entry declarations in order to emphasize the scope of the parameters. The formal part is then followed by the statements to be executed during the rendezvous. These are delimited by **do** and **end** and are the scope in which the parameters of the entry are accessible.

There are two possibilities for a rendezvous according to whether the calling task issues the calling statement such as

```
LINE_TO_CHAR.PUT_LINE(MY_LINE);
```

before or after a corresponding accept statement is reached by the called task. Whichever gets there first waits for the other. When the rendezvous is achieved, the appropriate parameters of the caller are passed to the called task (note that actual parameters are determined when the entry call is issued, not when the rendezvous occurs). The caller is then temporarily suspended until the called task completes the statements embraced by **do** ... **end**. Any **out** parameters are then passed back to the caller and finally both tasks again proceed independently of each other.

It should be observed that the rendezvous is named in one direction only. The calling task must know the name of the entry and this is specific to the called task. Thus the calling task must know the called task. The called task on the other hand will accept calls from any task. Thus we have a many-to-one pattern of communication. As a consequence of this, each entry potentially has a queue of tasks calling it. This queue is processed in a strictly first in first out manner and each rendezvous at an accept statement removes just one item from this queue.

The behavior of the process LINE_TO_CHAR should now be clear. It contains an internal buffer which may hold a line of characters. The task alternately fills the buffer by accepting a call of PUT_LINE and then empties it by accepting 80 successive calls of GET_CHAR. Calls of the entries can only be processed when the corresponding accept statement is reached. Thus many different tasks could be held up attempting to call PUT_LINE. They are only accepted one at a time in accordance with the groups of calls of GET_CHAR. Again note that the buffer may be emptied by several different tasks calling GET_CHAR. Indeed several tasks could be suspended on calls of GET_CHAR until a task issues a call of PUT_LINE.

It should be carefully observed that a task can only be in one queue at a time (and then only in it once). This is because a task can naturally only be calling one entry at a time.

This example could therefore be used to provide a simple buffering mechanism between a producer and consumer task thus (assuming the corresponding tasks have been initiated):

```
task CONSUME_CHAR;
task PRODUCE_LINE;

task body PRODUCE_LINE is
   use LINE_TO_CHAR;
   MY_LINE : LINE;
begin
   loop
      -- fill MY_LINE from somewhere
      PUT_LINE(MY_LINE);
   end loop;
end;

task body CONSUME_CHAR is
   use LINE_TO_CHAR;
   MY_CHAR : CHARACTER;
begin
   loop
      GET_CHAR(MY_CHAR);
      -- dispose of MY_CHAR
   end loop;
end;
```

In the task LINE_TO_CHAR there is only one accept statement corresponding to each entry. This need not necessarily be the case as later examples will show. Moreover if there are several accept statements corresponding to one entry then the bodies of the statements may differ. We see here a sharp distinction between entries and procedures. All calls of a procedure execute the same body whereas calls of entries need not. Entries are closely akin to coroutines in this respect.

As general programming practice the body of the entry between **do** and **end** should not contain unnecessary statements otherwise the calling task will be needlessly held up. As a consequence, it will often be the case that the **end** will follow the last statement which needs to access an entry parameter.

An accept statement may have no **do** ... **end** part. This will usually, but not necessarily, be the case when the entry has no parameters.

For example the following task implements a binary semaphore for protecting critical sections:

```
task SEMAPHORE is
   entry P;
   entry V;
end;

task body SEMAPHORE is
begin
   loop
      accept P;
      accept V;
   end loop;
end;
```

A critical section is then bracketed thus

```
P;
   -- critical section
V;
```

In this case the rendezvous merely provides synchronization and no data is transferred.

An entry can be declared in a task specification or in the outermost declarative part of a task body and is said to be owned by the task. In the simple examples we have met so far, the entries have all been declared in the visible part. The task READER_WRITER in the next section illustrates the use of a local entry.

There are constraints on the position of an accept statement to ensure that it is only executed by the task owning the corresponding entry. Firstly an accept statement may only be in the body of the task and not in an inner task. Secondly if an accept statement is in a procedure then that procedure may not be called directly, or indirectly, by a visible procedure or by the body of an inner task.

If an entry is renamed, it is renamed as a procedure. This preserves the uniform user interface. A minor distinction between entries and procedures is that it is not possible to have an entry function. Finally it should be remarked that it is possible to have families of entries. These are discussed in 11.2.8.

11.2.5 The Select Statement

The accept statement enables a task to wait for some event to happen and the happening of the event is in our notation indicated by the calling of the corresponding entry. To wait for several events all to have *happened* merely requires a sequence of accept statements. To wait for one only of several alternatives is not easy and for this purpose we introduce the select statement. As will become evident, the select statement has exceptional expressive power.

The select statement has some analogy with the case statement and in its simplest form allows one of several alternative accept statements to be obeyed.

As an example suppose we wish a variable to be accessible to many task but nevertheless wish to prevent more than one task from accessing it at the same time. Moreover suppose we wish to provide facilities to read the variable and to write a new value to it. The following task provides the entries READ and WRITE for this.

```
task PROTECTED_VARIABLE is
   entry READ (V : out ELEM);
   entry WRITE(E : in  ELEM);
end;

task body PROTECTED_VARIABLE is
   VARIABLE : ELEM;
begin
   loop
      select
         accept READ (V : out ELEM) do
            V := VARIABLE;
         end;
      or
         accept WRITE(E : in  ELEM) do
            VARIABLE := E;
         end;
      end select;
   end loop;
end PROTECTED_VARIABLE;
```

A call of READ copies the value of the variable into its parameter V. A call of WRITE copies the expression E passed as parameter into the variable.

The select statement allows the task to accept either READ or WRITE. On entry to the select statement if neither a READ nor a WRITE has been called, the task waits for the first of either and then obeys the appropriate accept statement. If one has already been called then that call is immediately accepted. If however both entries have already been called (obviously by two or more other tasks) then one of the alternatives is chosen in a completely random manner.

In the more general case each alternative may include a guarding condition following **when**. These conditions are all evaluated at the beginning of the select statement and only those alternatives whose guards are true are considered in the subsequent selection. An absent guard is of course considered to be true. If all guards are false so that no alternative can be considered, then it is an error (unless there is an else part as described later in this section) and the SELECT_ERROR exception is raised. An alternative whose guard is true (or absent) is said to be *open*. If the guard is false it is *closed*.

It should also be noted that each alternative may also include further statements following the rendezvous body of the accept. These additional statements are executed in the normal way after the rendezvous has been completed.

The following example of a bounded buffer illustrates the use of guards.

```
task BUFFERING is
   entry READ (V  : out ITEM);
   entry WRITE(E  : in  ITEM);
end;

task body BUFFERING is
   SIZE       : constant INTEGER := 10;
   BUFFER     : array (1 .. SIZE) of ITEM;
   INX, OUTX  : INTEGER range 1 .. SIZE := 1;
   COUNT      : INTEGER range 0 .. SIZE := 0;
begin
   loop
      select
         when COUNT < SIZE =>
            accept WRITE(E  : in  ITEM) do
               BUFFER(INX) := E;
            end;
            INX := INX mod SIZE + 1;
            COUNT := COUNT + 1;
      or
         when COUNT > 0 =>
            accept READ (V  : out ITEM) do
               V := BUFFER(OUTX);
            end;
            OUTX := OUTX mod SIZE + 1;
            COUNT := COUNT - 1;
      end select;
   end loop;
end BUFFERING;
```

The variables INX and OUTX index the ends of the currently used part of the buffer and COUNT indicates how many items are in the buffer. Note how obvious the guards are. A READ can only be accepted when the buffer is not empty and a WRITE can only be accepted when the buffer is not full. The reader is invited to compare the readability of the solution presented here with the example written in other languages in section 11.4.2.

It should be noted that the updating of the values of INX, OUTX and COUNT is not done within the rendezvous. This allows the calling task to continue as soon as possible.

The next example shows the use of local entries. It is an extension of the task PROTECTED_-VARIABLE described above and allows several tasks to READ simultaneously but only one to WRITE when no tasks are reading.

```
task READER_WRITER is
   procedure READ(V : out ELEM);
   entry WRITE(E : in ELEM);
end;

task body READER_WRITER is
   VARIABLE: ELEM;
   READERS : INTEGER := 0;
   entry START_READ;
   entry STOP_READ;

   procedure READ(V : out ELEM) is
   begin
      START_READ;
      V := VARIABLE;
      STOP_READ;
   end;

begin
   accept WRITE(E : in ELEM) do
      VARIABLE := E;
   end;

   loop
      select
         accept START_READ;
         READERS := READERS + 1;
      or
         accept STOP_READ;
         READERS := READERS - 1;
      or
         when READERS = 0 =>
            accept WRITE(E : in ELEM) do
               VARIABLE := E;
            end;
      end select;
   end loop;
end READER_WRITER;
```

In this example READ is a procedure and not an entry. However since entries are called in the same way as procedures the effective interface from the point of view of the caller remains unchanged. Of course the compiled calling code may be different but this need not concern the user.

This example also illustrates the use of more than one accept statement corresponding to the entry WRITE (in this particular example the bodies are the same but this need not be the case). It shows that a task can be viewed as a sort of *coroutine* where entry calls can achieve different actions depending on the current point of execution of the task.

We now consider a further elaboration of this example which gives a better distribution of priority between readers and writers. Normally writers have priority over readers and a new reader should not be permitted to start if there is a writer waiting.

However all waiting readers at the end of a write should have priority over the next writer. In order to program this strategy we use the attribute E'COUNT of an entry E which denotes the number of tasks waiting in the queue for the entry. The use of this attribute requires some care as explained below. We illustrate this point by means of two different formulations of this problem. The visible part of READER_WRITER remains unchanged. In the first formulation (not the better one), the declaration

```
N : INTEGER := 0;
```

is added to the declarative part of the body and the statement part now becomes as follows:

```
begin
   accept WRITE(E : in ELEM) do
      VARIABLE := E;
      N := START_READ'COUNT;
   end;

   loop
      select
         when WRITE'COUNT = 0 or N > 0 =>
            accept START_READ;
            READERS := READERS + 1;
            if N > 0 then
               N := N - 1;
            end if;
      or
            accept STOP_READ;
            READERS := READERS - 1;
      or
         when READERS = 0 and N = 0 =>
            accept WRITE(E : in ELEM) do
               VARIABLE := E;
               N := START_READ'COUNT;
            end;
      end select;
   end loop;
end;
```

In this formulation, N is the number of readers still waiting of those who were waiting when the previous write finished.

The *fifo* queue discipline is necessary for the correct working of this example. At the end of each write, the number of readers waiting is noted in N. A new reader is only accepted when there are no writers waiting unless some of the old readers are still to be served; hence the test of N in the guard of **accept** START_READ and the decrement of N in the body of START_READ. Similarly, the guard of **accept** WRITE ensures that a new writer is only served if there are no current readers or old readers still waiting.

The above formulation should be treated with caution. For consider what happens if one of the waiting readers is aborted while in the queue on the entry START_READ, and after the value of START_READ'COUNT has been assigned to N. The value of N will then become inconsistent and the next writer will be further delayed until a new reader arrives.

This illustrates a general danger with using the COUNT attribute in guards, since any task that has issued an entry call can receive an exception between the evaluation of COUNT and the execution of an accept statement based on the value of the COUNT. Note however that a guard of the form E'COUNT = 0 is never dangerous. No task was in the queue and so none can disappear.

We will now reformulate the above example avoiding the dangerous use of COUNT by introducing the else part of the select statement. A select statement may contain an else part following the various possibly guarded alternatives. The else part cannot be guarded. If all guards are false, or an immediate rendezvous is not possible, then the else part is obeyed. If there is an else part then a SELECT_ERROR cannot arise.

In the reformulated example, N is no longer required and the main loop now becomes as follows:

```
loop
   select
      when WRITE'COUNT = 0 =>   -- this is safe
         accept START_READ;
         READERS := READERS + 1;
   or
         accept STOP_READ;
         READERS := READERS - 1;
   or
      when READERS = 0 =>
         accept WRITE(E : in ELEM) do
            VARIABLE := E;
         end;
         loop
            select
               accept START_READ;
               READERS := READERS + 1;
            else
               exit;
            end select;
         end loop;
   end select;
end loop;
```

After accepting a WRITE the task loops accepting as many START_READs as can immediately be processed. Of course the behavior is marginally different but the general objective is satisfied. The loop ought also to follow the initial WRITE and so could conveniently be placed in a procedure.

There are constraints on the position of a select statement identical to those of the accept statement. These are of course necessarily imposed by the accept statements in the alternatives.

It is finally worth noting that the entries in the various alternatives need not be distinct (although they usually will). If two or more prove to be the same then the usual rule of random selection applies.

11.2.6 The Delay Statement

The delay statement postpones the execution of the task for at least the specified time interval.

```
delay 2.0*SECONDS;
```

The expression following delay represents the number of basic time units of the real time clock (on the object machine) for which the task is to be suspended. This expression will be of the predefined floating point type TIME. An integer type is not appropriate since the range of times to be accommodated will often exceed the range of integers on many object machines. The predefined constant SECONDS is the number of basic time units in one second and allows the expression to be written in a natural manner. The user can of course declare other appropriate constants. Thus

```
MINUTES : constant TIME := 60.0 * SECONDS;
```

It should be realized that the use of a floating point type for delays does not introduce any additional timing drift. The use of a simple delay in a loop such as

```
loop
   delay 1.0*MINUTES;
   ...
end loop;
```

does not cause the body of the loop to be executed every minute anyway because of the time taken to execute the statements in the loop. Accurate timing requires repeated reading of the clock and the use of a floating point type need not introduce any drift.

A delay statement may occur in place of an accept statement as the synchronization part of an alternative of a select statement and may have a guard in the usual way. Such a delay statement may be used to provide a time-out for the select statement. If no rendezvous has occurred within the specified interval then the statement list following the delay statement is executed. Of course if a rendezvous occurs before the interval has expired then the delay is cancelled and the select statement is executed normally.

As an example we can consider a task to drive a chain printer. If the printer does not receive any printing order for 10 seconds then the chain has to be stopped. Once it has stopped a further print request will cause it to restart but a 1 second delay must take place before printing commences.

```
task PRINTER_DRIVER is
   entry PRINT(L : LINE);
end;

task body PRINTER_DRIVER is
   CHAIN_GOING : BOOLEAN := FALSE;
   BUFFER      : LINE;
begin
   loop
      select
         accept PRINT(L : LINE) do
            BUFFER := L;
         end;
         if not CHAIN_GOING then
            -- start the chain
            delay 1.0*SECONDS;
            CHAIN_GOING := TRUE;
         end if;
         -- print the line
      or
         when CHAIN_GOING =>
            delay 10.0*SECONDS;
            -- stop the chain
            CHAIN_GOING := FALSE;
      end select;
   end loop;
end;
```

Of course a select statement may have several alternatives with a delay statement as the synchronization statement. This extends the general rule that the entries in the different alternatives need not be distinct. If there are several delays (with guard true) then the one with the shortest delay is chosen.

A select statement may not have alternatives commencing delay as well as an else part; the else part would always take precedence anyway. Moreover, at least one alternative must have an accept statement. This regularizes the constraint on the position of the select statement.

11.2.7 Interrupts

Hardware interrupts are simply handled by interpreting them as an external entry call. A representation specification is used to link the entry to the interrupt thus

```
for IO_DONE use at 4;
```

The value following *at* is interpreted in a machine dependent manner. For example, it could be a physical address, an index into a table of records, or a binary number representing encoded information.

The interrupt is processed when the task owning the entry performs a rendezvous by using a corresponding accept statement.

```
accept IO_DONE;
```

Multiple interrupts are queued as any other entry call is queued although there may be a system defined limit to the number of possible pending interrupts on a given entry.

The mechanism of masking interrupts is not visible to the user but is handled by the implementing software which connects the interrupts to the entry call. This approach enables the language to be reasonably machine independent in an otherwise awkward area.

An interrupt may return control information via an **in** parameter of the entry. Clearly an entry associated with an interrupt cannot have **out** or **in out** parameters.

11.2.8 Families of Tasks and Entries

The possibility of having families of tasks and entries has been mentioned above. A family is conceptually similar to an array but different terminology is used in order to maintain a clear distinction between program concepts such as tasks and data concepts such as integers. Furthermore the allocation strategy for families of tasks can be different from that of an array (see 11.4.6).

A natural use of a family of tasks occurs when there are several copies of a piece of physical equipment and a distinct but similar task is required to drive each one. Thus suppose we have 10 line printers and wish to drive each one by a distinct task such as PRINTER_DRIVER of section 11.2.6. The specification part would then become

```
task PRINTER_DRIVER (1 .. 10) is
   entry PRINT(L : LINE);
end;
```

The task body would remain the same. A member would be designated by appending a subscript in a manner analogous to an array component. Thus the sixth member would be initiated by

```
initiate PRINTER_DRIVER(6);
```

Alternatively the whole family would be initiated by

```
initiate PRINTER_DRIVER(1 .. 10);
```

In order to call an entry, the member of the family has to be indicated in full:

```
PRINTER_DRIVER(I).PRINT(MY_LINE);
```

It is often convenient to rename an entry in such circumstances so that calls are abbreviated. Thus:

```
procedure PRINT_6 renames PRINTER_DRIVER(6).PRINT;
```

A procedure in the visible part is called in a similar manner. Types, constants and exceptions however belong to the family as a whole and are denoted by just using the family name and not the member name.

Within the text of a family it is sometimes necessary to refer to the index of the current member. This can be done by the use of the INDEX attribute. Thus in the above case

```
PRINTER_DRIVER'INDEX
```

would give the index of the particular driver. It could, for example, be the case that the printers are controlled by a multiplexer. The printers could communicate with the multiplexer by calling its entries and passing their index as a parameter so that the multiplexer knows which printer it is dealing with at any time.

A family of tasks is an appropriate technique to use when the individual members correspond to physically autonomous pieces of equipment. We will now introduce families of entries and illustrate their use with a problem in which the individual physical items are not completely independent.

A family of entries is declared by adding a range specification to the entry name in the declaration. Thus

```
entry TRANSFER(1 .. 200)(D : DATA);
```

declares a family of 200 entries each of which has the parameter D.

A particular entry is called by the use of a subscript as expected

```
TRANSFER(I)(DATA_VALUE);
```

In the corresponding accept statement, the particular member has to be indicated by appending an actual index to the family name. It is then followed by the formal parameter list, if any, in the usual way.

```
accept TRANSFER(I)(D : DATA) do    ...    end TRANSFER;
```

Our example is that of scheduling a queue of requests for data transfers to or from a moving-head disk. In order to minimize head movement the requests are grouped into separate queues for each track and all the requests for a particular track are serviced together. It would be possible to consider each track as a separate physical entity demanding its own task. We would then use a family of tasks. However, the tracks are not independent. The disk can only be serving one track at a time and so the parallelism obtained by using many tasks is not necessary. Instead the transfers are handled by a single slave task with a family of entries. There is an entry for each track so that the queues are independent. A separate task controls the arm movement and the choice of track for the slave task.

```
task DISK_HEAD_SCHEDULER is
   type TRACK  is new INTEGER range 1 .. 200;
   type DATA   is ... -- other parameters of transfer
   procedure TRANSMIT(TN : TRACK; D : DATA);
end;

task body DISK_HEAD_SCHEDULER is
   type DIRECTION is (UP, DOWN);
   INVERSE: constant array (UP .. DOWN) of DIRECTION :=
                     (UP => DOWN, DOWN => UP);
   STEP   : constant array (UP .. DOWN) of INTEGER range -1 .. 1 :=
                     (UP => 1, DOWN => -1);
   WAITING: array (TRACK'FIRST .. TRACK'LAST) of INTEGER := (TRACK'FIRST .. TRACK'LAST => 0);
   COUNT  : array (UP .. DOWN) of INTEGER := (UP .. DOWN => 0);
   MOVE   : DIRECTION := DOWN;
   ARM_POSITION : TRACK := 1;

   entry SIGN_IN(T : TRACK);
   entry FIND_TRACK(REQUESTS : out INTEGER; TRACK_NO : out TRACK);

   task TRACK_MANAGER is
      entry TRANSFER(TRACK'FIRST .. TRACK'LAST)(D : DATA);
   end;

   procedure TRANSMIT(TN : TRACK; D : DATA) is
   begin
      SIGN_IN(TN);
      TRACK_MANAGER.TRANSFER(TN)(D);
   end;

   task body TRACK_MANAGER is
      NO_OF_REQUESTS: INTEGER;
      CURRENT_TRACK  : TRACK;
   begin
      loop
         FIND_TRACK(NO_OF_REQUESTS, CURRENT_TRACK);
         while NO_OF_REQUESTS > 0 loop
            accept TRANSFER(CURRENT_TRACK)(D : DATA) do
               -- do actual I/O
               NO_OF_REQUESTS := NO_OF_REQUESTS - 1;
            end TRANSFER;
         end loop;
      end loop;
   end TRACK_MANAGER;
```

```
begin  -- DISK_HEAD_SCHEDULER
  initiate TRACK_MANAGER;
  loop
    select
      when COUNT(UP) + COUNT(DOWN) > 0 =>
        accept FIND_TRACK(REQUESTS: out INTEGER; TRACK_NO: out TRACK) do
          if COUNT(MOVE) = 0 then
            MOVE := INVERSE(MOVE);
          else
            ARM_POSITION := ARM_POSITION + STEP(MOVE);
          end if;
          while WAITING(ARM_POSITION) = 0 loop
            ARM_POSITION := ARM_POSITION + STEP(MOVE);
          end loop;
          COUNT(MOVE) := COUNT(MOVE) - WAITING(ARM_POSITION);
          REQUESTS  := WAITING(ARM_POSITION);
          TRACK_NO  := ARM_POSITION;
          WAITING(ARM_POSITION) := 0;
        end FIND_TRACK;
    or
      accept SIGN_IN(T : TRACK) do
        if T < ARM_POSITION then
          COUNT(DOWN) := COUNT(DOWN) + 1;
        elsif T > ARM_POSITION then
          COUNT(UP) := COUNT(UP) + 1;
        else
          COUNT(INVERSE(MOVE)) := COUNT(INVERSE(MOVE)) + 1;
        end if;
        WAITING(T) := WAITING(T) + 1;
      end SIGN_IN;
    end select;
  end loop;
end DISK_HEAD_SCHEDULER;
```

The user indicates his requests by calling the procedure TRANSMIT. This in turn calls the entry SIGN_IN in the main task which records the request and then the user waits on the call of TRANSFER until the slave task TRACK_MANAGER is ready to perform transfers on the track concerned.

The slave task TRACK_MANAGER calls the entry FIND_TRACK in order to determine which track should be handled next. DISK_HEAD_SCHEDULER only honors the call when there are requests outstanding (COUNT(UP)+COUNT(DOWN)>0). If there are requests outstanding, an extended rendezvous occurs during which the arm is moved and the data transferred to TRACK_MANAGER.

Note the accept statement within TRACK_MANAGER which references the member CURRENT_TRACK of the family TRANSFER and so finally deals with the user who has been waiting in TRANSMIT.

It should be pointed out that the example given is purely illustrative. No genuine disk head scheduler would need to be so heavily engineered. A perfectly adequate solution is to allow only two calls to track manager at a time and to sort these into the more efficient order. If a disk queue frequently exceeds two items then the system is grossly overloaded anyway and elaborate scheduling is unlikely to help.

11.2.9 Generic Tasks

The final concept to be described is that of a generic task. A full description of generics is given in chapter 12 of the Reference Manual and we merely describe here a simple example which illustrates the need for and use of generic tasks.

We saw in section 11.2.4 how a simple task could implement a binary semaphore for protecting a critical region. The main shortcoming of the example was that it only implemented a single semaphore and did not provide a mechanism whereby the user could declare as many semaphores as he wished. By making the task generic this difficulty can be overcome and we illustrate it by the allied example of a signal.

A simple signal has two operations, SEND and WAIT. One task may SEND a signal and another may WAIT for it. When a signal is sent only one waiting task is released. A signal is remembered until a task waits but repeated sends are ignored. A solution is as follows:

```
generic task SIGNAL is
   entry SEND;
   entry WAIT;
end SIGNAL;

task body SIGNAL is
   HAS_OCCURRED : BOOLEAN := FALSE;
begin
   loop
      select
         accept SEND;
         HAS_OCCURRED := TRUE;
      or
         when HAS_OCCURRED =>
            accept WAIT;
            HAS_OCCURRED := FALSE;
      end select;
   end loop;
end;
```

Within the user a particular signal is declared thus:

```
task MY_SIGNAL is new SIGNAL;
```

The calls of WAIT and SEND then appear as

```
MY_SIGNAL.WAIT;
MY_SIGNAL.SEND;
```

Signals and semaphores are predefined generic tasks. This should enable an implementation to make optimal use of any synchronization facilities provided by the machine or underlying system. For example, if semaphores are supplied by a given hardware, calls to the P and V operations can be mapped directly on the corresponding hardware primitives. In such a case the package body should only be considered as a semantic description of the corresponding construct.

A variation on the normal signal is the so-called *passing* signal. In this case the signal is not remembered and a task must wait for a later occurrence of the signal. A signal sent when no one is waiting is completely lost. A solution is

```
task body PASSING_SIGNAL is
begin
   loop
      accept SEND;
      select
         accept WAIT;
      else
         null;
      end select;
   end loop;
end PASSING_SIGNAL;
```

11.2.10 Scheduling

The key to designing parallel tasks in the Green language lies in the realization that queues are associated with entries and only entries and that such queues are handled in a strictly first in first out manner. The example of the DISK_HEAD_SCHEDULER showed how a family of entries could be used to fragment a queue into subqueues. The *fifo* nature of the entry queue might be thought to be a severe constraint in cases where some requests may be of high priority and also in cases where later similar requests could be satisfied even though earlier ones had to wait.

The handling of requests with priorities is easily achieved by the use of separate entries for each level. A family can conveniently be used for that purpose. The following example illustrates an approach suitable for a small number of levels.

```
task CONTROL is
   type LEVEL is (URGENT, MEDIUM, LOW);
   entry REQUEST (LEVEL'FIRST .. LEVEL'LAST)(D : DATA);
end;

task body CONTROL is
   ...
   select
      accept REQUEST(URGENT)(D : DATA) do
         ...
      end;
   or when (REQUEST(URGENT)'COUNT = 0) =>
      accept REQUEST(MEDIUM)(D : DATA) do
         ...
      end;
   or when (REQUEST(URGENT)'COUNT = 0) and (REQUEST(MEDIUM)'COUNT = 0) =>
      accept REQUEST(LOW)(D : DATA) do
         ...
      end;
   end select
   ...
end CONTROL;
```

The use of the COUNT attribute in the above example is quite safe.

For a larger number of levels, a different approach may be more appropriate as is illustrated below:

```
task CONTROL is
   subtype LEVEL is INTEGER range 1 .. 50;
   procedure REQUEST (L : LEVEL; D : DATA);
end;

task body CONTROL is
   entry SIGN_IN  (L : LEVEL);
   entry PERFORM (LEVEL'FIRST .. LEVEL'LAST)(D : DATA);
   PENDING : array (LEVEL'FIRST ..LEVEL'LAST) of INTEGER :=
                                  (LEVEL'FIRST .. LEVEL'LAST => 0);
   TOTAL    : INTEGER := 0;

   procedure REQUEST(L : LEVEL; D : DATA) is
   begin
      SIGN_IN(L);
      PERFORM(L)(D);
   end;

begin
   loop
      if TOTAL = 0 then
         -- no request to be served:  wait if necessary
         accept SIGN_IN(L : LEVEL) do
            PENDING(L) := PENDING(L) + 1;
            TOTAL := 1;
         end SIGN_IN;
      end if;
      loop  -- accept any pending SIGN_IN call without waiting
         select
            accept SIGN_IN(L : LEVEL) do
               PENDING(L) := PENDING(L) + 1;
               TOTAL := TOTAL + 1;
            end SIGN_IN;
         else
            exit;
         end select;
      end loop;

      for I in reverse LEVEL'FIRST .. LEVEL'LAST loop
         if PENDING(I) > 0 then
            accept PERFORM(I)(D : DATA) do
               -- satisfy the request of highest level
            end;
            PENDING(I) := PENDING(I) - 1;
            TOTAL := TOTAL - 1;
            exit;  -- restart main loop in order to accept new requests
         end if;
      end loop;
   end loop;
end CONTROL;
```

In order to service a request, a call to SIGN_IN must first be accepted, and its occurrence recorded in the global counter TOTAL, and the appropriate PENDING counter. In a second step, the appropriate entry of the family PERFORM must be accepted. CONTROL proceeds by

- waiting for the first SIGN_IN if all previous requests have been serviced;
- accepting all pending calls to SIGN_IN;
- executing the request with the highest priority;
- going back to the beginning of the loop to take care of any call to SIGN_IN that has arrived in the meantime.

We will now illustrate a very general mechanism which in effect allows the items in a queue on a simple entry to be processed in an arbitrary order. The example is of a controller for the allocation of groups of items from a set of resources.

```
task MULTI_RESOURCE_CONTROL is
   type RESOURCE is (A, B, C, D, E, F, G, H, I, J, K);
   type RESOURCE_SET is array (A .. K) of BOOLEAN;
   procedure RESERVE(GROUP : RESOURCE_SET);
   entry RELEASE(GROUP : RESOURCE_SET);
end;

task body MULTI_RESOURCE_CONTROL is
   EMPTY : constant RESOURCE_SET := (A .. K => FALSE);
   USED  : RESOURCE_SET := EMPTY;

   entry      FIRST  (ASKED : RESOURCE_SET; OK : out BOOLEAN);
   entry      AGAIN  (ASKED : RESOURCE_SET; OK : out BOOLEAN);
   procedure  TRY    (ASKED : RESOURCE_SET; OK : out BOOLEAN);

   procedure  RESERVE (GROUP : RESOURCE_SET) is
      POSSIBLE : BOOLEAN;
   begin
      FIRST(GROUP,POSSIBLE);
      while not POSSIBLE loop  -- if at first you don't succeed, try again
         AGAIN(GROUP,POSSIBLE);
      end loop;
   end;

   procedure TRY(ASKED : RESOURCE_SET; OK : out BOOLEAN) is
   begin
      if (USED and ASKED) = EMPTY then
         USED := USED or ASKED;
         OK := TRUE;     -- allocation successful
      else
         OK := FALSE;    -- not possible, try again later
      end if;
   end;
```

```
begin  -- MULTI_RESOURCE_CONTROL
   loop
      select
         accept FIRST(ASKED : RESOURCE_SET; OK : out BOOLEAN) do
            TRY(ASKED,OK);
         end;
      or
         accept RELEASE(GROUP : RESOURCE_SET) do
            USED := USED and not GROUP;
         end;
         for I in 1 .. AGAIN'COUNT loop
            select
               accept AGAIN(ASKED : RESOURCE_SET; OK : out BOOLEAN) do
                  TRY(ASKED,OK);
               end;
            else
               exit;
            end select;
         end loop;
      end select;
   end loop;
end MULTI_RESOURCE_CONTROL;
```

The user requests and obtains an arbitrary group of resources by calling the procedure RESERVE and returns resources by calling the entry RELEASE. The procedure RESERVE makes an immediate attempt to acquire the resources by calling the entry FIRST. If they are not all available, OK is returned false and the request is queued by calling AGAIN. It should be noted that FIRST is always honored promptly (except when the controller is busy with RELEASE) whereas AGAIN is only considered when a RELEASE occurs. Thus all requests which cannot be satisfied immediately are placed on the AGAIN queue. It is important that these requests are not serviced on a FIFO basis but that when some resources are released the requests in the queue that can be fully satisfied should be honored. The technique is to scan the queue by doing a rendezvous with AGAIN and to allow each user (in RESERVE) to place itself back on the queue if it cannot get the resources it requires. In order that each user should have only one retry the loop is controlled by AGAIN'COUNT and it is important that AGAIN'COUNT is only evaluated once at the start of the loop.

It should be observed that users may reenter the AGAIN queue in a slightly different order because of the underlying task scheduling strategy. This does not affect the validity of the algorithm. Note also that the accept statement is placed inside a select statement with an else part that terminates the loop. This ensures that if a waiting task is aborted then the system does not deadlock.

The technique is general but does require the use of a special protocol for calling FIRST and AGAIN, although this is hidden by making these entries local and by enclosing the entry calls in the procedure RESERVE. It may appear surprising that the bodies of FIRST and AGAIN are identical, but FIRST is always necessary since the user must be allowed the resources immediately if they are available.

11.3 Example: A Radar Track Management Package

This example shows the use of packages and tasks to realize a complex real-time system, such as radar surveillance.

The TRACK_MANAGEMENT package introduces the abstract notion of a track. Several tracks can coexist. A current position and a current speed vector in a two-dimensional space are associated with each track. The position is updated regularly from the value of the speed. Both the position and speed can be modified externally, or examined. The restrictions are that those values should not be read while they are being changed. We thus have a classical reader-writer problem.

```
package TRACK_MANAGEMENT is
   type TRACK_INFO is
      record
         X, Y    : MILES;                    -- global type
         VX, VY  : MILES_PER_SECOND;         -- global type
         T : TIME;
      end record;

   type TRACK_ID is private;

   function    CREATE_TRACK  (INIT : in TRACK_INFO) return TRACK_ID;
   procedure   KILL_TRACK    (T  : in TRACK_ID);
   function    READ_TRACK    (T  : in TRACK_ID) return TRACK_INFO;
   procedure   CHANGE_TRACK  (T  : in TRACK_ID; D : in TRACK_INFO);
   NO_MORE_TRACKS, ILLEGAL_TRACK : exception;
private
   MAX_TRACK : constant INTEGER := 512;
   subtype TRACK_RANGE  is INTEGER range 0 .. MAX_TRACK;
   subtype NAME_TYPE    is LONG_INTEGER;
   type TRACK_ID is
      record
         INDEX : TRACK_RANGE;
         UNIQUE_NAME : NAME_TYPE;
      end record;
end TRACK_MANAGEMENT;

package body TRACK_MANAGEMENT is
   NULL_TRACK  : constant TRACK_ID := (0, 0);
   TRACK_NAME  : array (1 .. MAX_TRACK) of NAME_TYPE := (1 .. MAX_TRACK => 0);
   -- this is a table indicating, for each track, the unique name currently assigned to it.
   LAST_NAME   : NAME_TYPE := 0;  -- the last unique-name used.

   procedure CHECK_TRACK (T : in TRACK_ID);

   task TRACK (1..MAX_TRACK) is
      procedure READ(I : out TRACK_INFO);
      entry CHANGE(I : in TRACK_INFO );
      entry INITIALIZE(I : in TRACK_INFO);
      entry KILL;
   end TRACK;
```

```
task TRACK_CONTROL is
   entry CREATE_TRACK(ID : out TRACK_ID);
   entry KILL_TRACK(T : in TRACK_ID);
end TRACK_CONTROL;

procedure CHECK_TRACK (T : in TRACK_ID) is
   -- to ensure the validity of a track value:
   -- * it has a positive index,
   -- *  it has the same unique name as that known to the system,
   -- * it is still active.
begin
   if T.INDEX = 0
      or else TRACK_NAME(T.INDEX) /= T.UNIQUE_NAME
      or else not TRACK(T.INDEX)'ACTIVE then
      raise ILLEGAL_TRACK;
   end if;
end CHECK_TRACK;

function CREATE_TRACK (INIT : in TRACK_INFO) return TRACK_ID is
   NEW_TRACK : TRACK_ID;
begin
   TRACK_CONTROL.CREATE_TRACK(NEW_TRACK);
   initiate TRACK(NEW_TRACK.INDEX);
   TRACK(NEW_TRACK.INDEX).INITIALIZE(INIT);
   return NEW_TRACK;
end CREATE_TRACK;

procedure KILL_TRACK(T : in TRACK_ID) is
begin
   CHECK_TRACK(T);
   TRACK(T.INDEX).KILL;
   TRACK_CONTROL.KILL_TRACK(T);
end KILL_TRACK;

function READ_TRACK (T : in TRACK_ID) return TRACK_INFO is
   I : TRACK_INFO;
begin
   CHECK_TRACK(T);
   TRACK(T.INDEX).READ(I);
   return I;
exception
   when TASKING_ERROR => raise ILLEGAL_TRACK;
end READ_TRACK;

procedure CHANGE_TRACK ( T : in TRACK_ID; D : in TRACK_INFO) is
begin
   CHECK_TRACK(T);
   TRACK(T.INDEX).CHANGE(D);
exception
   when TASKING_ERROR => raise ILLEGAL_TRACK;
end CHANGE_TRACK;
```

```
task body TRACK is
   DATA      : TRACK_INFO;
   READERS   : INTEGER := 0;
   entry START_READ;
   entry STOP_READ;

   procedure READ (I : out TRACK_INFO) is
   begin
      START_READ;
      I := DATA;
      STOP_READ;
   end READ;

   procedure UPDATE_POSITION is
      NEW_TIME   : TIME := SYSTEM'CLOCK;
      DELTA_TIME : TIME := NEW_TIME - DATA.T;
   begin
      DATA.X := DATA.X  + DELTA_TIME*DATA.VX;
      DATA.Y := DATA.Y  + DELTA_TIME*DATA.VY;
      DATA.T := NEW_TIME;
   end UPDATE_POSITION;

begin     -- body of TRACK;
   accept INITIALIZE (I : in TRACK_INFO) do
      DATA := I;
   end INITIALIZE;
   UPDATE_POSITION;

   loop
      select
         when CHANGE'COUNT = 0 and KILL'COUNT = 0 =>
            accept START_READ;
            READERS := READERS + 1;
      or
         when READERS > 0 =>
            accept STOP_READ;
            READERS := READERS - 1;
      or
         when READERS = 0 and KILL'COUNT = 0 =>
            accept CHANGE (I : in TRACK_INFO) do
               DATA := I;
            end CHANGE;
            UPDATE_POSITION;
      or
         accept KILL;
         exit;
      or
         delay 0.10*SECONDS;
         UPDATE_POSITION;
      end select;
   end loop;
end TRACK;
```

```
task body TRACK_CONTROL is
   function FIND_TRACK return TRACK_RANGE is
   begin
      for I in 1 .. MAX_TRACK loop
         if TRACK_NAME(I) = 0 then
            return I;
         end if;
      end loop;
      raise NO_MORE_TRACKS;
   end FIND_TRACK;

begin      -- body of TRACK_CONTROL
   loop
      begin
         select
            accept CREATE_TRACK(ID : out TRACK_ID) do
               ID.INDEX := FIND_TRACK;
               if LAST_NAME  = NAME_TYPE'LAST then
                  LAST_NAME  := 1;
               else
                  LAST_NAME := LAST_NAME + 1;
               end if;
               TRACK_NAME(ID.INDEX) := LAST_NAME;
               ID.UNIQUE_NAME := LAST_NAME;
            end CREATE_TRACK;
         or
            accept KILL_TRACK(T : in TRACK_ID) do
               TRACK_NAME(T) := 0;
            end KILL_TRACK;
         end select;
      exception
         when NO_MORE_TRACKS => null;
      end;
   end loop;
end TRACK_CONTROL;

begin      -- body of TRACK_MANAGEMENT
   initiate TRACK_CONTROL;
end TRACK_MANAGEMENT;
```

Tracks are manipulated by external agents through the operations CREATE_TRACK (to start a new track, with an initial value), READ_TRACK (to obtain the current position on a track), CHANGE_TRACK (to modify the track data) and KILL_TRACK (to release the track).

All tracks are independent. This is achieved by associating a particular task from a family to a newly created track. The global management of the pool of tasks is achieved by the TRACK_CONTROL task. Note that the TRACK tasks act as *servers*, in the sense that an activation of one task corresponds to one track, but the same task can represent different tracks in different successive activations.

In order to preserve some integrity in the way tracks are used (for example, to ensure that a reference to a track is not that of an obsolete activation), a unique name is associated with each active track. This unique name is a long integer which is incremented at each track creation, and recorded in the track identification. It acts as a sort of password, in that, for each active track, the system keeps the unique name currently associated to it in the array TRACK_NAME. This one is checked against that contained in the track identification. Using a LONG_INTEGER for unique names should guarantee that the same name is not reused before a reasonable period of time.

For each track, the policy is that a track cannot be changed while it is being read, that no new reader should be accepted if a change is to be made, and that termination should have priority over both reading and writing. One of the consequences is that a track termination is possible while some readers are still accessing it. However, any reader, or any waiting writer will receive a TASKING_ERROR exception when this happens.

11.4 Rationale for the Design of the Tasking Facilities

The section starts by briefly surveying some of the more important and older real-time primitives and their shortcomings. It then considers the concept of rendezvous and shows how this concept has influenced the design of the Green tasking facilities. It finally discusses the decisions involved in the design of the different tasking features.

11.4.1 Early Primitives

The understanding of algorithmic sequential processes is based upon that of the evaluation of arithmetic and Boolean expressions whose axioms have been well understood for centuries. However, there is no mathematical tradition upon which we can draw in order to help us to understand the behavior of cooperating sequential processes. As a consequence it has been difficult to decide whether a particular set of real-time primitives is good or not. Many sets can be implemented in terms of each other but their relative primitiveness is often hard to perceive.

Broadly speaking the primitives (or perhaps the applications) can be divided into two categories. The first enables common data or common code to be protected from multiple usage. The second enables one task to send a message to another; this includes the degenerate case of a signal which can be thought of as a message with no contents.

One of the oldest and best known primitive sets is the boolean semaphore described by Dijkstra [Di 68]. This consists of the two operators P and V acting on a semaphore S which takes two values, busy and free (or equivalently true and false). The behavior of the operations is:

P(S) If S is busy the task is suspended until S becomes free. If S is free then it is set busy and the task proceeds.

V(S) S is set free. If there are tasks held up on a P(S) operation then one of them is allowed to proceed.

Semaphores can be used to protect data by surrounding code which accesses the data by matched calls thus:

```
P(S)                                P(S)
   -- access data                      -- access data
V(S)                                V(S)

task A                              task B
```

Semaphores can also be used to signal happenings. One task waits by calling P, the other signals by calling V.

```
P(S)  wait for B                    V(S)  signal to A

task A                              task B
```

Semaphores can therefore be used both for protection and signalling. They also have the merit of being primitives that are both simple to describe and easy to understand. What then are their disadvantages? Briefly the problem is that for all but the simplest applications, the programming of semaphores is difficult. Programs using semaphores exhibit similar symptoms to unstructured programs using gotos. They are hard to write, understand, prove and maintain.

More specifically, typical problems are:

- One can jump around a call of P and therefore accidentally access unprotected data.
- One can jump around a call of V and accidentally leave the semaphore busy so that the system deadlocks.
- One can forget to use them.
- It is not possible to program an alternative action if a semaphore is found to be busy when attempting P.
- It is not possible to wait for one of several semaphores to be free.
- Semaphores are often visible to tasks which need not access them.

An extended form of semaphore is the integer semaphore. In this case the value is an integer rather than a boolean. It is particularly useful for allowing a limited number of tasks to have access to a resource. Nevertheless it has been shown that the integer semaphore can be programmed in terms of boolean semaphores and so in practice it is only marginally more useful.

Closely related to the semaphore is the signal or event. There are variations but a typical definition would be that an event E has two states, set and unset, and the following operations upon it:

WAIT(E) — If E has not been set then the task is suspended until the event is set. If E has been set then it is unset and the task is allowed to proceed.

SEND(E) — E is set. If there are tasks waiting for E then one of them is allowed to proceed.

Clearly such an event is isomorphic to the Boolean semaphore. The difference lies perhaps in the intended use. Semaphores are associated with data protection and events with indicating that something has happened. There are variations in which several events are remembered. But in all forms, events suffer from the same structuring problems as semaphores.

Various other primitives have been proposed in order to overcome the structuring difficulties of semaphores and events. Newer proposals have been put forth, but they usually tackle only one of the application areas distinguished above (data protection and signalling). In this respect they are somewhat unbalanced.

The critical section has been proposed as a syntactic form equivalent to a bracketed pair of P and V operations. This prevents goto statements from bypassing one of the operations and hence overcomes some of the difficulties of semaphores. A further form, the conditional critical section, allows an alternative action to be performed if the resource represented is busy.

Critical sections do not seem to have been successful. They only solve the exclusion problem and need to be complemented with a signalling mechanism; this does not lead to the unification sought by language designers.

Many forms of message switching system have been implemented in order to give improved solutions to the signalling problem (see [BH 70, 73]). Typically they enable messages to be sent between tasks and allow the source or destination of the message to be optionally specified. They therefore give added protection by preventing unauthorized access to messages.

Perhaps the biggest disadvantages of message systems is the need for a sizable message controller. Message systems also seem to be of an ad-hoc nature with a nonobvious set of parameters. Moreover they do not easily solve exclusion problems because of the high overhead involved.

A significant step forward was the monitor first described by Brinch Hansen [BH 73,75] and by Hoare [Ho 74]. This includes the facilities of the critical section and when combined with events (as in Modula), gives a reasonable solution to problems such as the bounded buffer. The monitor solves the exclusion problem but not the message problem. Indeed the signals in Modula still suffer from all the structuring problems of semaphores.

11.4.2 The Rendezvous Concept

Another line of approach to mutual exclusion and synchronization was introduced in early computer science by Conway [Co 63] with the notion of *coroutine*, the first definition of a high level synchronization mechanism. One of the important concepts introduced by Conway (and maybe forgotten later) is that synchronization and data transmission are two inseparable activities. Two parallel tasks need to be synchronized to exchange information, thereafter they resume their respective activities; this synchronization is known as a *rendezvous*. Two recent papers by Hoare [Ho 78] and Brinch Hansen [BH 78] propose a rethink of parallel processing in terms of this concept of rendezvous and have strongly influenced the design of the Green language.

The difficult problem that arises here is one of making tasks known to each other. Tasks have names that identify them unambiguously. Should these names be used by tasks to synchronize with each other, or should there exist a further entity that makes both candidates for synchronization known to each other by reference to some common *channel*? These two solutions are extreme forms of *symmetric* communication; either each communicating task has full knowledge of its colleague, or it has no information at all. Both solutions appear in the literature: [Ho 78] and [Ka 74].

We rejected the channel solution in this design in order to avoid an additional language concept and the dual connection mechanism that it requires. The solution adopted in Green, although closer to the solution proposed by Hoare is *asymmetric*: one of the two communicating tasks knows the name of the other one and names it explicitly; the second task just knows that it expects some external interaction.

In order to justify the asymmetry, we first summarize the symmetric proposal developed by Hoare and embedded in a language which has become known as CSP (Communicating Sequential Processes). Communication between tasks is seen as synchronized input-output. One task outputs data which the other inputs and both tasks rendezvous during the transfer - that is the first to arrive at its input or output statement waits for the other and they both then execute the I/O statements together (or apparently together) before proceeding independently. Each task names the other in the transfer. The transfer can be thought of as an assignment split into two parts with the left side in one task and the right side in the other. The source and destination must be compatible for assignment.

As an example we will consider a task BUFFERING to smooth variations in the speed of output of items by a producer task and input by a consumer task (given in section 11.2.5). The program is as follows:

```
BUFFERING ::
   buffer : (1 .. 10) item;
   inx, outx, count : integer;
   inx := 1; outx := 1; count := 0;
      *  [count < 10; producer?buffer(inx) ->
            inx := inx mod 10 + 1; count := count + 1
      [] count > 0; consumer?more() ->
            consumer!buffer(outx);
            outx := outx mod 10 + 1; count := count - 1
      ]
```

The key language statements in this example are:

```
X ? Y   Input Y from task X
X ! Y   Output Y to task X
```

On each iteration the guards "count < 10" and "count > 0" are evaluated. If both guards are true then calls from either the consumer or producer are acceptable and the first such call will be waited for; if both have already made such a call and are therefore themselves waiting then a nondeterministic choice will be made; if only one has made a call then obviously that call is taken. If, however, only one guard is true then only the corresponding call can be accepted and the other task will wait until the buffer is partially filled or emptied as the case may be. In this example both guards cannot be false and so the iterative process continues indefinitely.

In the producer case the statement

```
producer ? buffer(inx)
```

moves the item into the buffer directly. In the consumer case the statement

```
consumer ? more()
```

indicates that the consumer is ready and a subsequent

```
consumer ! buffer(out)
```

actually does the transfer. The producer task therefore contains statements such as

```
BUFFERING ! X
```

whereas the consumer task has pairs such as

```
BUFFERING ! more();
BUFFERING ? X
```

Note that *more*() denotes a structured value with no components and is used here as a signal.

As can be seen the program is readable although perhaps presented in a terse style by traditional high level language standards. One of the main problems with CSP is that a (one-to-one) named correspondence is required. Because of this symmetry, it is not possible to program a library routine to provide resources to arbitrary users.

The preliminary definition of the Green language was similar to CSP in its semantics except that naming was only one-sided. Tasks can be characterized as services and as users. A user certainly needs to know the name of the service it is requesting. On the other hand a service need not know the names of the users. Because of this asymmetry it became possible to program the library routine. As a consequence there can be queues of waiting tasks associated with each request. On each successful rendezvous just one waiting task is served. In preliminary Green the transfer locations were known as *boxes* and were restricted to being simple names. Boxes had a direction associated with them by analogy with procedure parameters. The notion of *box*, as a typed input or output register was also similar to the notion of *port* introduced in [Ba 70] and [Wa 72].

The buffering example in preliminary Green was as follows:

```
path BUFFERING (MORE : in box; IN_ITEM : in box ITEM; OUT_ITEM : out box ITEM);
  BUFFER : array (1 .. 10) of ITEM;
  INX, OUTX, COUNT : INTEGER;
begin
  INX := 1; OUTX := 1; COUNT := 0;
  loop
    select MORE ! IN_ITEM of
      when COUNT < 10 receive IN_ITEM =>
        BUFFER(INX) := IN_ITEM;
        INX := INX mod 10 + 1; COUNT := COUNT+1;
      when COUNT > 0 receive MORE =>
        OUT_ITEM := BUFFER(OUTX);
        OUTX := OUTX mod 10 + 1;  COUNT := COUNT-1;
        send OUT_ITEM;
    end select;
  end loop;
end;
```

The call in the producer task was

```
connect BUFFERING(IN_ITEM := X);
```

and in the consumer we had

```
connect BUFFERING(MORE);
connect BUFFERING(OUT_ITEM =: X);
```

The interpretation was similar to CSP but the syntax was more traditional. The select statement indicated the multiple choice and is embedded in a loop statement. The guards are Boolean expressions following the reserved word **when**. An absent guard was taken to be true. The guards were followed by a synchronization statement, *receive* or *send* referring to a box of type **in** or **out** respectively, the text following => was obeyed after the corresponding rendezvous was completed.

The main trouble with the facilities in both CSP and preliminary Green is that a double interaction is required for the consumer. This means that the two calls really need to be *wrapped up* into a single procedure in order to give a clean interface. It is worth comparing the above with the same example written in Modula using monitors as follows:

```
interface module buffering;
  define put, get;
  var buffer : array 1 .. 10 of item;
    inx, outx, count : integer;
    nonfull, nonempty : signal;

  procedure put (x : item);
  begin
    if count = 10 then wait(nonfull) end;
    buffer [inx] := x;
    inx := inx mod 10 + 1; count := count + 1;
    send(nonempty)
  end put;

  procedure get (var x : item);
  begin
    if count = 0 then wait(nonempty) end;
    x := buffer [outx];
    outx := outx mod 10 + 1; count := count - 1;
    send(nonfull)
  end get;
begin
  inx := 1; outx := 1; count := 0
end buffering;
```

The producer and consumer process move the items by calls such as

```
put(x)    and    get(x)
```

respectively. This is satisfactory but the internal behavior of the monitor is not nearly as clear as in CSP and Green.

Perhaps the most important point about CSP and Green is that they offer mechanisms which are applicable to both data protection and signalling. Earlier attempts to develop features at a higher level than semaphores or events (such as message systems and monitors) seemed to solve only one problem and by offering an unbalanced solution were not clearly better than the original simple primitives.

An important concept introduced in this final form of the Green language is the notion of the *extended* rendezvous. This notion is a major jump to a higher level of abstraction. In the case of the task BUFFERING this overcomes the need for the double rendezvous with the consumer. This is seen by comparing the example in section 11.2.5 with that in 11.4.2. Thus we now have

```
BUFFERING.READ(X)
```

rather than

```
connect BUFFERING(MORE);
connect BUFFERING(OUT_ITEM =: X);
```

This also illustrates the procedural form of entry call as opposed to the specialized connect statement. As we have seen this enables a constant external interface to be presented even if a change of solution demands that a procedure be replaced by an entry or vice versa.

The extended rendezvous is more disciplined since it ensures that the same task performs the interaction throughout. It should also be observed that the rendezvous mechanism is more disciplined than a monitor since the accept statements appear inside a context (e.g. following a guard) from which information can be deduced, thereby facilitating both understanding and proof.

The introduction of entries rather than boxes leads naturally to the unification of tasks and packages. A task encapsulates a collection of entries in the same way as a package encapsulates data and procedures. Moreover there is a strong analogy between the specification part in which the entries are specified and the body containing the sequence controlling the critical actions.

However, this unification has its limits since it has been necessary to disallow variables and modules in the visible part. This is both for methodological reasons (the variable would appear to be controlled by the task although it would not) and because of the cost of preventing access to variables of an inactive task and of implementing access if the system is distributed.

The general applicability of the rendezvous concept has been confirmed by its use in other examples. This concept is well adapted to distributed systems (communication is achieved by entry calls, exchanged data is passed via parameters). From a more theoretical viewpoint, it is interesting to note that path expressions [CH 74] can be shown to be easily expressible in terms of rendezvous primitives.

11.4.3 Task Declarations

The declarative part of a task body is elaborated each time the task is initiated. This is necessary in order to ensure that on each initiation the internal starting conditions are identical; local variables must be reinitialized. Elaboration on initiation is also indicated by a consideration of task families which are discussed below.

The declarative part of a task specification on the other hand is elaborated only once when the task is declared. This ensures that types exported from the visible part preserve compatibility from one activation to another. It also indicates that types exported from a family belong to the family as a whole. The difference in time of elaboration of specification and body is a further argument for the separation of their texts.

11.4.4 Initiation and Termination

The fork and join mechanism with automatic initiation on scope entry and suspension of the parent until exit was rejected in favor of a more explicit initiation mechanism for various reasons.

Explicit initiation gives more control in repairing a faulty system. It makes more visible any orderly start-up. It is cleaner for repetitive use and easier to program a watchdog (see 12.4.1).

Multiple initiation is necessary so that two cooperating tasks can be started together and can then each assume that the other is active.

There is a rule that a task cannot exit a scope until all tasks declared in that scope have terminated. *This is simply because of the need to have orderly control of storage allocation since a task could* otherwise access the data of enclosing units.

An explicit abort statement seems necessary, perhaps reluctantly, in order to control a wayward process. Raising a FAILURE exception is not adequate if the failed task has for one reason or another programmed its *last wishes* in an uncooperative manner. Unconditional termination is the only consistent semantics that can be given to the abort statement, if it is to be useful. For obvious scoping reasons this implies aborting all descendant tasks. If an aborted task is engaged in a rendezvous, then this rendezvous must be abandoned as well and its partner must be informed (via an exception) so that it may know whether or not the rendezvous has been completed. Of course, if the partner is also being aborted in the same action, then no exception need be raised. An aborted task also has to be removed from any entry queue on which it may have been placed as a result of obeying an entry call. This does not raise any exception but demands care in the use of the COUNT attribute. Note that the FAILURE exception and the abort statement can be used in conjunction as in the following termination sequence:

```
if T'ACTIVE then
   raise T.FAILURE;
   delay 20.0*SECONDS;
   abort T;
end if;
```

This sequence allows the task T a period of 20 seconds in which to terminate itself gracefully before the abort statement is applied. If T has already terminated, the abort statement will have no effect.

The abort statement is provided for emergency use only. Its overuse could severely hinder program understanding and validation.

11.4.5 Examples of Task Hierarchy

It is most important that the reader should understand the distinction between the parent of a task, the initiator of a task and the task embracing the declaration of a task. In most cases these will be the same task but they can be distinct. This section contains (perhaps somewhat contrived) examples to illustrate the possibiliites. Consider

```
task body Z is
   task W is
      ...
   end;

   task body W is
      ...
   end W;

   task X is
      procedure P;
   end;

   task body X is
      task Y is
         ...
      end;

      task body Y is
         ...
      end Y;

      procedure P is
      begin
         initiate Y, W;
      end;
      ...
   end X;
begin
   initiate X;
   X.P;
end Z;
```

Task Z calls procedure P in X which results in the initiation of tasks Y and W. The parent of Y is however X and not Z. The parent of W is Z. Note that X is guaranteed to be active since otherwise P could not have been called. The procedure P could equally have been called by X.

Note that in the above example, if X had been a package and not a task, the parents of both Y and W would have been Z. Furthermore suppose that P contains a local task V.

```
procedure P is
   task V is
   ...
   end V;

begin
   initiate V;
end;
```

Then if procedure P is called from Z , then Z is the parent of V whereas if P is called from task X then X is the parent of V.

11.4.6 Task Families

The task family is a useful and necessary concept for several reasons.

It is a common requirement to have many similar tasks active simultaneously. If the number is at all large, then the declaration of several different tasks (or even their generic instantiation) could be tedious. Moreover these mechanisms are constrained to occur only at scope entry. (Thoughts of recursive procedures containing generic instantiations are not fruitful).

The task family provides a model of several tasks and the individual tasks are designated by the members of the family. The number of possible active tasks in the family is of course determined by the range in the task declaration and can be evaluated on scope entry. Moreover the explicit initiate statement enables the number of members actually used to be closely controlled.

It is important to notice that types, constants, and exceptions declared in the visible part belong to the family as a whole. This is desirable so that an exported type is equivalent over all members of the family. On the other hand, entries and procedures belong naturally enough to the members of the family since they are the means of communication with the individual members. Observe that this distinction does not exist with generics. Each instantiation gives rise to a new version of any visible type. Task families are significantly different from generics in this respect.

11.4.7 Implementation of Task Creation

The implementation of task creation can be either dynamic or static. Dynamic creation means that storage for a task is allocated when the task is initiated. Static creation means that storage for the task is reserved when the task declaration is elaborated. This choice is normally left to the implementation since the optimal strategy is not the same on all machines. However it is possible to influence the choice of the translator by the pragmas

```
pragma CREATION (STATIC);
pragma CREATION (DYNAMIC);
```

These considerations apply to all tasks but are particularly relevant for task families.

For a small family (one with a small total storage requirement) it may be more efficient to do a static allocation in a manner analogous to a *cactus* stack. This effect can be achieved by the above pragma used in conjunction with a length specification indicating the maximum space required by each task for its data and for the subprograms that it calls. This may be especially useful for a family that remains permanently active.

More generally, however, the discrete range of a task family should just be considered as defining the *allowable* indices for members of the family, but it does not mean that all members have to be allocated at scope entry. Consider the following example:

```
type NAME is new INTEGER range 1 .. 1000;

task PROCESS (NAME'FIRST .. NAME'LAST) is
   pragma CREATION (DYNAMIC);
   -- specifications of the entries of PROCESS
end PROCESS;

A, B, C : NAME;
TABLE : array (1 .. 10) of NAME;
```

Variables of type NAME can be viewed as identifying individual tasks of the family with names of the form PROCESS(A), PROCESS(B), and so on. In addition values of type NAME can be remembered in arrays such as TABLE, or in record components. Task activations can thus be designated by the family name (PROCESS) and by a value of type NAME.

An activation of a task of the PROCESS family is created by the execution of an initiate statement. In this example creation may be dynamic because of the corresponding pragma. In an application the role of assigning unique names for tasks could be delegated to a procedure SET_NAME and the creation of the task C would be achieved by the sequence

```
SET_NAME (C);
initiate PROCESS (C);
```

Other languages such as Tartan [SHW 78] have tried to achieve a similar effect by the introduction of additional language constructs. For example Tartan introduces a specific data type for designating activations of task, both in a named form and in an anonymous manner. Thus in the named form, the declaration of A, B, C would appear as

```
var A, B, C : activation of PROCESS;
```

This named form is safe but it is clear that it does not achieve more than what the Green form does without introducing the additional concept of activation name. The anonymous form of activation names provided by Tartan corresponds to what other languages have called task variables. For example with

```
var ANY_TASK : actname;
```

the variable ANY_TASK would be able to refer to any possible task (for example to DISK_HEAD_SCHEDULER or to LINE_TO_CHAR). This means that it would be possible to perform entry calls such as

```
ANY_TASK.TRANSMIT (T, D);
ANY_TASK.GET_CHAR (C);
```

We considered this possibility in this design and rejected it as being too low level and incompatible with the reliability required in embedded computer systems. In terms of complexity also, such *untyped* task variables raise the same issues as formal procedures in Algol 60 (how do we establish that an entry call is legal for such untyped task variables?). Finally, it would not have been consistent to introduce formal tasks in this manner in a language that otherwise does not provide formal procedures and procedure variables.

Note that the most important use of *anonymous* task names can safely be covered by existing language concepts. It corresponds to the case of a server needing the ability to recognize its customers. In such a case restricted private types provide a facility whereby a key may be created and handed to a task on its first request. On later requests the task shows the key, thus enabling the server to recognize the owner. The private type mechanism prevents forgery of the key. There is a risk that keys will get reused in the future, since a normal mechanism would be to represent each new key as the next integer. This risk is no more than that associated with remembered task activation variables and indeed can be minimized to any required degree by using a key composed of a record of several integers. Thus using just two 16 bit words a new key can be issued every second for 136 years without duplication. The TRACK_MANAGER package of section 11.3 provides an example of the use of such keys.

11.4.8 Procedure and Entry Interface

The possibility of having procedures as well as entries in the visible part of a task does produce great methodological power. For example we have seen in the task READER_WRITER in 11.2.5 how the procedure READ forces the entries START_READ and STOP_READ to be called in the correct order. Indeed without the procedure there would be no guarantee that pairs of calls were matched at all.

Variables have been disallowed from the visible part of tasks for methodological reasons (they would appear controlled by the task although they are not) and because of difficulties with preventing their access when the task is not active. However a function can be used to read a local variable of a task in a controlled manner.

It should be realized that the caller executes a procedure himself whereas an accept statement is executed by the callee on the caller's behalf.

A problem with allowing procedures is that it essentially gives the caller the ability to probe around in the body of the callee. A major consequence is the constraint necessarily imposed on the position of the accept statement (accept statements cannot be executed by such procedures). Note finally that on distributed systems (where tasks do not share a common store) communication by procedure calls may be disallowed, all communication being achieved by entry calls. For such entry calls, parameter passing would be implemented by copying.

11.4.9 Accept Statement

The rationale behind the accept statement and entry call is simply to provide a rendezvous. In some applications it is necessary that a rendezvous be achieved whereas in others it is important for the caller not to be held up. It is much more difficult to program a rendezvous in terms of non-rendezvous primitives than vice versa. Hence the rendezvous has been chosen as the natural primitive.

It is noted that calls are accepted in simple order of arrival. The alternative of making the order depend on some parameter of the call was considered and rejected because of the difficulty with implementation which could severely penalize the simple user. As has been demonstrated in the examples, it is possible to program different strategies when necessary.

11.4.10 Select Statement

It may be felt surprising that the alternatives in a select need not refer to different entries. One motivation here is the fact that if several alternatives are open, one of them is chosen at random and there is hence no reason to disallow the same entry in two alternatives. Another motivation is the existence of families of entries. If E(I) and E(J) were two entries and they had to be different, then a tedious runtime check would be necessary. The rule thus allows different actions to be programmed in a simple way on the same entry but according to different guards.

Note that the guards are all evaluated at the start of the select statement only. The alternative semantics of evaluating a guard only when an entry is called was considered and rejected. The problem concerns the indivisibility of evaluating the guard and accepting the call together. One could not afford to make the guard evaluation indivisible and so it would be possible for the calling task to be aborted during the guard evaluation. This would cause havoc if the guard proved to be true.

Guard evaluation at the start of the select statement could be criticized on the grounds that the value of a guard may be changed by another task before an alternative is chosen. This is not a good argument since even if the guard were evaluated when the corresponding alternative is chosen, there is no guarantee that it might not be immediately changed. In either case there is a danger with the use of asynchronously modifiable guards (such as those containing COUNT and CLOCK, etc). Note that in practice most guards are local to the task containing the select statement. In addition they are most often very simple. Consequently several optimizations of guard evaluation are possible.

The rule for choosing one of the open alternatives has been stated to be at random. This should be treated in a statistical sense. It should not be possible for a program to detect the algorithm used. It would certainly be unwise to assume for instance that the alternatives were taken in some order. If a uniform strategy is desired, then it must be programmed by using appropriate guarding conditions.

The need for the else part has been adequately demonstrated by earlier examples. It should be observed that the select statement allows a server to choose between different accept statements. There is no corresponding mechanism for a caller to choose between the first of several calls. This is because of a fundamental design decision: a task can only be on at most one queue at a time. The main motivation for this decision is simplicity and efficiency of the implementation.

11.5 Implementation Considerations

A possible implementation of the Green tasking facilities is outlined in this section. This description should certainly not be considered as the only possible implementation, inasmuch as it often sacrifices efficiency for the sake of a simpler presentation.

11.5.1 General Description

The proposed implementation is oriented towards machine configurations consisting of one or more processors accessing a common memory. It is also assumed that processors can be interrupted, in particular by a clock.

An implementation of the tasking facilities must consider the four following points: storage allocation, organization of queues, implementation of scheduling, and implementation of the select statement. The choices made in this implementation are presented below.

11.5.1.1 Storage Allocation

The storage necessary for a task includes that needed for control information, for local objects, and for activation of subprograms called by the task. All this information is found in what is called a task activation record. These task activation records are allocated in the space of the enclosing task, when the declaration of the local task is elaborated. This strategy corresponds to a static allocation, as the storage is allocated as long as the task can be named, independently of its initiations. The alternative approach is to allocate storage only at task initiation.

11.5.1.2 Organization of Queues

The runtime system must maintain certain queues of tasks. The only queues actually needed are the *ready queue* (containing the tasks that can be run if a processor is available), the entry queues (containing all the tasks that have performed a call to the entry considered), and the delay queue (containing all the tasks that have executed a delay statement which has not yet expired).

The entry queues and the delay queue are simple linked lists, whereas the ready queue has a more elaborate structure: a small number of priorities is assumed (for example 5), and the ready queue consists of separate linked lists, one per priority.

Except for the delay queue, a task is entered at the tail of a queue, and is generally removed from the head. This policy reflects the first in, first out scheduling imposed by the language. The delay queue is always sorted by completion date, that is, the task with the earliest completion time is first in the queue.

11.5.1.3 Implementation of Scheduling

when a process must be scheduled, the ready queues are examined in decreasing order of priority. The first process on the first nonempty queue is selected and removed from its queue. Deciding when the scheduler should be invoked is a choice left to the implementer. For instance a scheduler may guarantee a certain percentage of process time to each priority level, use time slicing, etc. These strategies do not affect the language concepts. They could vary from one implementation to another. Alternative scheduling strategies may also be provided for the same implementation.

In the implementation outlined here, a simple policy is used: the scheduler is invoked either when a processor becomes available, or when a new task is entered in the ready queue. This corresponds to the following situations:

- initiation of a task;
- termination of a running task;
- entry call;
- reaching an accept statement for which no call has been issued, or a select statement for which there is no possible alternative for immediate execution;
- termination of a rendezvous;
- execution of a delay statement;
- expiration of a delay;
- reception of an interrupt awaited by a task.

An additional scheduling decision must be made when a task alters its priority.

11.5.1.4 Implementation of the Select Statement

Various schemes can be devised to choose an alternative in a select statement. In order to simplify the presentation, we assume that a random number generator is used to select an alternative when several alternatives are open.

We shall now proceed with the detailed description of the information needed at runtime, and of the various operations associated with the tasking facilities.

11.5.2 Information Needed at Runtime

Four categories of runtime information are considered:

- information associated with a particular task
- information associated with a unit that contains a task declaration
- information associated with an entry
- global information about the whole system.

11.5.2.1 Information Associated with a Task

This information can be stored in the task activation record. It includes:

(a) The Context of the Task:

This is the information required to reactivate a suspended task. It includes the value of the program counter, a pointer to the current subprogram activation when the task was suspended, and depending on the machine, the value of other registers.

(b) The Dynamic Link

This is a pointer to the activation record of the unit that contained the task declaration.

(c) The State of the Task

This is a boolean flag indicating if the task is *active*, that is, if it has been initiated, and has not terminated.

(d) The Task Priority

(e) The Next Task Pointer.

This is used to chain together the tasks that are on the same queue. Since a given task can only be in at most one queue at a given time, only one such field is actually needed.

(f) The Entry Descriptors (see section 11.5.2.3)

A pointer (*first entry*) is found at a fixed offset in the task activation record, containing a reference to the first entry descriptor. The number of entries is also stored.

(g) The Pending Exception Field

This identifies an exception that may have been received by a suspended task. The rules of the language guarantee that only one such field is needed.

Additional storage may be needed for recording the task's cumulative processing time and for delay information, if the task body contains a reference to the CLOCK attribute or a delay statement. The delay information merely consists of the time at which the task is to be reactivated. Since in the case of multiple delays in a select statement, only one delay is actually executed, the value of the program counter is sufficient to indicate where processing should resume.

11.5.2.2 *Units Containing Task Declarations*

A unit in which tasks are declared must contain an *Inner Task Counter,* which gives the number of tasks declared in the unit that are active. When a local task is initiated, this counter is incremented; it is decremented upon termination of such a task. This counter is needed for an efficient implementation of the rule that a scope cannot be left before all local tasks have terminated.

A boolean variable is needed to indicate if the execution of the unit has reached its final end but is not yet terminated because of the existence of active local tasks.

A unit containing declarations of task families whose bounds are not known at compile time must also contain indirect references to the task activation records of each family. For all other tasks, the location of the task activation record within the current unit can be determined at compilation or link edition time.

11.5.2.3 Information Associated with an Entry

Each entry has an entry descriptor, at a fixed offset in the task activation record. This descriptor consists of

(a) The State of the Entry

The state of an entry is *open* if the task in which it is declared is ready to accept a rendezvous for this entry, and no call has been issued. It is *closed* otherwise.

(b) The Entry Queue

All tasks that are waiting on a call to the entry are placed in this queue in order of arrival. The entry descriptor contains a pointer to the first and last task activation records on that queue.

(c) The Transfer Address

When a task is suspended on an accept statement (waiting for the entry to be called) the value of the program counter, saved in the context of the task, indicates where processing should resume. However, if the accept clause is nested in a select statement, the program counter is no longer sufficient to provide this information. When the entry is opened, the address corresponding to a particular accept statement is saved in the transfer address field of the entry descriptor.

It is assumed that all entry descriptors are stored contiguously in the task activation record.

11.5.2.4 Global Information

As described earlier, the runtime maintains a ready queue and a delay queue for tasks that are awaiting a processor, or the expiration of a particular delay. Queue pointers similar to those used for entries (head and tail pointers) are found at a fixed address in the system. Additional information is gathered in:

(a) The Processor Table

This table indicates, for each processor, the location of the task activation record of the task currently being executed on the processor and the value of the real-time clock when the processor started running this task. .

(b) The Interrupt Tables

For each interrupt, the table indicates the entry linked to the interrupt. If the entry has no parameter, then the entry queue is actually used as a counter which records the number of times the corresponding interrupt has been received and not accepted by the task. If the entry has parameters, their value is saved in a special storage area, and linked together in a queue.

11.5.3 Runtime Routines

The routines described here are invoked in different circumstances. Some of them should be uninterruptible, as they alter the global information of the system.

11.5.3.1 Task Declaration

For the simple implementation outlined here a static allocation scheme is used (as for the pragma CREATION(STATIC)). A task activation record is allocated when the unit containing the task declaration is entered within the storage space of the enclosing task.

The initialization of the dynamic link can be done at this point, and the task state is set to *inactive* (these operations need not be performed in uninterruptible mode).

11.5.3.2 Task Initiation

An initiate statement can be performed in three stages:

(a) Suspend the initiating task (saving its context).

(b) For each task to be initiated, set the task priority to that of the initiating task, initialize the task context, close all its entries, and chain the task to either the initiating one (if it is the first in the initiate statement), or to the previously initiated task.

(c) For each initiated task, increment the inner task counter of its declaring unit (found through the dynamic link), and set the task state to active. Finally, place the initiating task and all initiated tasks in the ready queue corresponding to their common priority, and invoke the scheduler.

It should be noted that on a single processor, it is not necessary to invoke the scheduler, as the initiating task can be immediately resumed. However, in the case of multiple processors, the newly initiated tasks may have a higher priority than some of the tasks running on other processors, thus justifying a scheduling decision.

These operations should be uninterruptible. This is mandatory for steps (a) and (c). If step (b) is to be made interruptible, then each initiated task must be set to a state (such as *being initiated*) during step (a), in order to prevent another task from attempting another initiation in parallel.

11.5.3.3 Task Termination

Normal termination operations consist of

(a) Setting the task state to inactive.

(b) Decrementing the inner task counter of the declaring unit. If, as a result, the counter becomes null, and the declaring unit has reached its end (the end-of-unit flag being true), then the outer task must be resumed, by entering it in the appropriate ready queue. The corresponding task activation record can be found by following the dynamic chain of activations until a task activation is encountered.

(c) Entering the scheduler to find a new task to be run.

All these operations must be uninterruptible because both the task state and the inner task counter could be accessed by other tasks.

11.5.3.4 Entry Call

When a task S calls an entry E of a task T:

(a) The actual parameters are evaluated; their values are then available in the activation record of S.

(b) The task S is suspended (the program counter and current activation pointer are saved) and placed on the entry queue of E.

(c) If E is open, then all other entries of T are closed; the transfer address for E is copied in the program counter, and T is entered in the appropriate ready list. If T was in the delay list, it is removed; if T was the first task in the delay queue, the timer must be reset to a value corresponding to the delay time of the next task in the delay queue.

(d) A new task is chosen by the scheduler.

Steps (b) through (d) should be uninterruptible. Note that in step (b), the *first entry* field of the task is used to find all the entry descriptors that are stored contiguously.

11.5.3.5 Reaching an Accept Statement

When reaching an accept statement for an entry E, two cases are possible:

(1) If the waiting queue of E is empty, then:

   (a) The state of the entry is set to open.

   (b) The current value of the program counter for the task is saved in the transfer address of E.

   (c) The task is suspended (not placed in any queue) and a new task is scheduled.

(2) If the waiting queue of E is not empty, then a rendezvous can be accepted immediately: the first task in the waiting queue of E is removed from that queue. A link to it is saved on the stack of T, and the statements corresponding to the rendezvous can be executed. The same sequence of actions should be performed when the task is reactivated by an entry call, after having been suspended in case (1).

Operations described in (1) are uninterruptible. In (2), only the removal of the caller from the waiting queue is uninterruptible.

11.5.3.6 End of a Rendezvous

At the end of a rendezvous, the called task must place the calling task back in the appropriate ready queue (this is, of course, uninterruptible). The called task is then suspended, and the scheduler is invoked (to deal with the case of a caller having a higher priority).

11.5.3.7 Delay Statement

When a simple delay statement is executed, or inside a select statement if no other alternative is available for immediate execution, the wake-up time is computed by adding the value of the delay expression to the value of the real-time clock. This time is saved in the delay descriptor of the task. The task is entered in the delay queue at the place corresponding to the wake-up time. If it is added at the head of the queue, then the value of the timer (considered as a register that is decremented at each clock tick, and that raises an interrupt when its value reaches 0) must be changed to the new delay value. Adding the task to the delay queue and updating the timer must be uninterruptible.

11.5.3.8 Occurrence of a Timer Interrupt

When a timer interrupt is received, the delay queue must be scanned. All tasks having delay values not greater than the current time are removed from the delay queue, and entered in the ready queues corresponding to their respective priorities. The new value of the timer is determined from the smallest delay value of all tasks remaining on the delay queue. The scheduler must then be invoked.

All these operations should be uninterruptible.

11.5.3.9 Select Statement

We describe here a fairly straightforward implementation of the select statement. More efficient approaches can be devised, in the absence of side effects in guarding expressions (the usual case).

All guards are first evaluated (in their textual order), and their values collected in a bit vector. If a guard is true and the corresponding alternative starts with a delay, the delay expression is evaluated and compared with the value stored in the delay field of the task. The smallest of the two values is kept in this field, and a reference to the particular branch is saved as the task's program counter value.

If no guard is true, and the select statement does not have an else part, then a SELECT_ERROR is raised. Otherwise, in uninterruptible mode, the entries corresponding to true guards are examined.

Three cases are possible:

(1) More than one such entry has a non-empty waiting queue: a random number generator is called to choose one of the possible alternatives.

(2) Exactly one entry has a non-empty waiting queue: it is selected for immediate execution.

(3) All entry queues with true guards are empty. If an else clause is present, it is executed. If not, the state of all entries with true guards is set to open, and the address of the code corresponding to each alternative is saved in the entry descriptors. If the delay value of the task is not null, the task is entered in the delay queue, as described in 11.5.3.7, and in the other case, the task is merely suspended.

In cases (1) and (2) execution of the task proceeds with the rendezvous corresponding to the selected alternative. When the task is suspended, the scheduler must be invoked.

11.5.3.10 Alternative Implementations of Select Statements

Alternative implementations of random selection are possible. Two such methods are sketched below: the *select-marker* method is particularly efficient in the presence of guarding conditions without side-effects (the most likely case), because it avoids reevaluation of some of the guards. The *order of arrival* method takes into account the time elapsed since the arrival of the tasks on the various queues.

The select-marker method

For each select statement in a task, an integer index is reserved in the task activation record and initialized to some positive value in the range 1 .. N, where N is the number of alternatives in the select statement.

When a select statement is reached, alternatives are considered in turn, starting with the one whose position corresponds to the index. The guard is evaluated. If it is false, or if the entry queue is empty, the index is incremented by 1 (modulo N) and the next alternative is considered. If tne guard is true and the entry queue non-empty, then the alternative is immediately selected for execution (the index is also incremented). At the next execution of the select statement, alternatives will thus be considered starting with the one immediately following the last selected alternative.

If all alternatives have been considered, and all those with true guards had empty queues at the time they were examined, then the queues must be reexamined in an uninterruptible mode. If some of the queues have become nonempty, then one of the corresponding alternatives must be selected (e.g. the one closest to the index value), and the index appropriately updated.

The order of arrival method

In this method, each task containing entry declarations has a global *entry call counter*, to hold sufficiently large integer values. This counter is initialized to 0. Each task also contains a variable of the same size that holds the *order of arrival*.

Whenever a task T issues a call to an entry of a task U and the entry is closed, the value of the entry call counter of U is incremented by 1 and the new value is copied as the order-of-arrival of task T.

When a select statement is to be executed, the guards are evaluated and the first task on each nonempty entry queue is examined. The one with the smallest order-of-arrival is selected for execution.

11.5.3.11 *Treatment of Interrupts*

When an interrupt is received, the following interrupt routine is executed:

The interrupt table indicates which entry is linked to the interrupt (it is assumed that if an interrupt does not correspond to any entry, it will be masked).

In the case of a parameterless entry, the entry queue is replaced by a counter that is incremented by 1. If the entry was open, all entries of the task are closed and the task is placed in the ready queue (after updating the program counter with the value stored in the entry descriptor). The interrupt can then be cleared and the scheduler entered.

In the case of an entry with parameters (**in** only), the values are read at specified addresses. Space is obtained from a special storage area to copy these values, and this space is attached to the queue of the entry.

The nature of the interrupts that could be received during execution of this routine and what would happen to interrupts that are masked is machine and implementation dependent and thus not described here.

12. Exception Handling

12.1 Introduction

The ability to handle error situations is essential for reliability of real time systems. In many cases, they must be designed as systems which should never halt. This definitely requires an ability to handle situations which, although rare, are quite likely to happen given enough time.

This subject of exception handling has received considerable attention in recent years, and several language formulations of exception handling features have been proposed. For a presentation of these facilities the reader is referred to the extensive accounts given in [Go 75] and [Le 77]. The solutions proposed differ mainly on the level of generality they give to the concept of exception.

A first family of solutions tends to consider exception handling as a normal programming technique for events that are infrequent, but not necessarily errors. This viewpoint has been followed in [LMS 74], [Go 75], [PW 76], [Le 77] and [GS 77]. It means that when an exception occurs it is treated by an exception handler, and then control may return to the point where the exception occurred. It also means that exception handling may be used to perform some repair actions and to continue normal execution thereafter.

A second family of proposals tends to restrict exceptions to events that can be considered (in some sense) as errors or, at least as terminating conditions. This means that when an exception occurs in a given program unit, its execution will be terminated. Control will be passed to an exception handler but will never return to the point where the exception occurred. The handler may decide to restart the same sequence of actions under better conditions, but it will do so by a different invocation of these actions, not a simple resumption.

This second family of solutions includes recovery blocks [HLMR 74, Ra 75] and a recent proposal by Bron, Fokkinga and De Hass [BFH 76]. The Steelman requirements for exception handling clearly require a solution in this second family, since they specify that the occurrence of an exception should cause transfer to a handler without completing the elaboration of the declaration or the execution of the statement where the exception occurred.

Naturally, what is considered as an error is rather subjective, and the ability of a handler to reinvoke a subprogram that raised an exception will permit the use of exception handling for making repairs and for the treatment of rare events. The problem domains that can be addressed by the two families of solutions are hence comparable, but they require different programming styles and different underlying mechanisms.

The exception handling facility provided in the Green language belongs to this second family. It provides a facility for local termination upon detection of errors. It has been inspired by the Bron proposal and has some similarities with the Bliss signal enable construct [DEC 74].

Our goal in this design has been to define an exception handling facility involving a minimum number of concepts and well integrated with the facilities for parallel processing. A major goal was to have a solution compatible with an axiomatic definition, hence permitting program verification and mechanical program optimization.

The discussion of exception handling starts by an overall presentation followed by examples illustrating the main classes of uses. The interactions between exceptions and parallel processing are then presented and we conclude by a discussion of several technical issues.

12.2 Presentation of the Proposal for Exception Handling

The definition of an exception handling facility must provide answers to the following questions:

- How are exceptions declared?
- What are exception handlers and in which part of a program can they appear?
- How are exceptions raised?
- Which handler gets executed when an exception is raised?
- How can an exception be reraised?
- How can exceptions be suppressed?

We next examine these different questions in the case of sequential programs. The case of parallel tasks is discussed in section 12.4.

12.2.1 Declaration of Exceptions

The name associated with an exception condition must be declared. The form of an exception declaration is shown by the following example:

```
SINGULAR : exception;
```

Conceptually we may view the set of all exception declarations appearing in a program as forming the equivalent of a distributed declaration of the literals of an enumeration type, say *exception_condition*, whose values may only be mentioned in exception handlers and in raise statements. Thus the above declaration has the meaning that SINGULAR is one of the possible exception conditions.

Declarations for predefined exceptions such as INDEX_ERROR, RANGE_ERROR, TASKING_ERROR, OVERFLOW, etc. are provided in the predefined environment.

12.2.2 Exception Handlers

Exception handlers are the sections of the program to which control is passed when exceptions occur. Each exception handler has the form of a sequence of statements prefixed by the reserved word **when** followed by the names of the exceptions serviced by the handler considered.

Exception handlers may only appear at the end of a subprogram body, module body, or block after the reserved word **exception**. As an example the following block contains a single handler that services the exception SINGULAR;

```
begin
   -- sequence of statements
exception
   when SINGULAR =>
      PUT("matrix is singular");
end;
```

A handler started by **when others** services all exceptions for which no explicit handler is given in the same program unit. Note finally, that any sequence of statements with which an exception handler is needed can always be made into a block.

12.2.3 The Raise Statement

There are two possible reasons for an exception to be raised in a given program unit. It may either be explicitly raised by a raise statement or, as we will explain later, it may be *propagated* by subprograms (including operators) and blocks executed by the program unit considered.

The main form of raise statement includes the reserved word **raise** and the name of the exception that is raised:

```
raise SINGULAR;
raise FILE_HANDLING.END_OF_FILE;
```

The name of the exception must of course be visible at the point of the raise statement. It may have the form of a selected component as in the above case of the exception END_OF_FILE declared in the package FILE_HANDLING.

12.2.4 Association of Handlers with Exceptions

We next examine the question of finding which handler gets executed when a given exception is raised. Note that if a program unit contains a raise statement for a given exception, it does not necessarily contain a handler for that exception.

For example, in the procedure P given below, both the procedures P and R provide a handler for SINGULAR and have no explicit raise statement for that exception. On the contrary, the procedure Q contains an explicit raise statement for SINGULAR but provides no handler for that exception.

```
procedure P is
   SINGULAR : exception;

   procedure Q is
   begin
      ...
      if DETERMINANT = 0 then
         raise SINGULAR;
      end if;
      ...
   end Q;

   procedure R is
   begin
      ... Q ...
   exception
      when SINGULAR =>
         -- second handler for SINGULAR
   end R;

begin -- P
   ... R ... Q ...
exception
   when SINGULAR =>
      -- first handler for SINGULAR
end P;
```

When an exception is raised within a program unit, we may be in either of the two following situations:

(a) The currently executing subprogram or block has no handler for the exception;

The execution of the subprogram is terminated and the same exception is implicitly raised at the point of call of the subprogram. Similarly the execution of the block is terminated and the same exception is implicitly raised after the block in the enclosing program unit. In both cases we say that the exception is *propagated*.

(b) A handler has been provided for the exception:

The actions following the point where the exception is raised are skipped and the execution of the handler terminates the execution of the current program unit.

In the above example, if the exception SINGULAR is raised during the execution of Q called by R, the execution of Q will be terminated, since no handler is provided for SINGULAR within Q. This exception is then *propagated* to the caller: it is raised within R at the point of call of Q and serviced by the second handler. The execution of this handler terminates the execution of R and the exception is not further propagated. For P, this call of R therefore appears as a normal call. Note that the first handler for SINGULAR, that of P, would be executed if the exception were raised by the execution of Q corresponding to its direct call within P.

With this definition of exception handling, the effect of a subprogram is normally completed by the sequence of statements of its body; when an exception occurs, it may be completed by a corresponding handler, if present.

The initialization part of a package body acts as a procedure implicitly called by the package for its initialization. This applies also for exceptions. A handler in a package body acts as a handler in a procedure. In the absence of a handler, an exception is propagated to the program unit containing the package declaration. The case of task bodies is discussed in Section 12.4.

Having explained the concept of exception propagation, it should now be clear that there is no conceptual difference between the predefined exceptions and the exceptions declared by the user. Predefined exceptions are exceptions that can be propagated by the basic operations of the language such as indexing, accessing a value, and the arithmetic operations. As an example DIVIDE_ERROR is an exception that may be propagated by the (hardware supplied) operation of division.

12.2.5 Reraising an Exception

Within a handler, the exception that caused transfer to the handler may be reraised by a normal raise statement (mentioning its name) or by a raise statement of the form

```
raise;
```

In either case, the effect of reraising the same (or another) exception within a handler is to terminate the current program unit and to propagate the corresponding exception (except for tasks as explained in section 12.4).

The abbreviated form for reraising an exception is especially useful in the case of a handler for **others**. Thus, such a handler can be used to perform some general clean-up actions, such as undoing possible side-effects, before raising the same exception again. This is made possible by the fact that the exception is left anonymous both in the handler prefix and in the reraise statement.

12.2.6 Suppressing an Exception

It is possible to indicate to the translator that the detection of some predefined exceptions need not be provided within a given program unit. This is achieved by inserting a pragma such as

```
pragma SUPPRESS(NO_VALUE_ERROR);
```

As a result the translator may omit any extra code to check that a variable has been initialized. Note that this pragma is not imperative, and does not mean that the designated exception will not be raised in the program unit, as it may be raised explicitly, be propagated from a subprogram where it has not been suppressed, or simply because the translator did not inhibit the check. The latter is likely to be the case if hardware detection of the exception is available.

This facility is especially useful for some predefined exceptions, such as ASSERT_ERROR or NO_-VALUE_ERROR, as their detection may be expensive unless aided by special hardware.

12.2.7 Order of Exceptions

A translator may choose to evaluate the constituent terms of an expression in any order that is consistent with the precedence properties of the operators, and with the parentheses. As a consequence, the order in which exceptions might occur in the evaluation of an expression is not guaranteed by the language. The formal semantics of the language only defines the value of an expression whose evaluation does not raise any exception.

12.3 Examples

Several examples presenting typical uses of exception handling are discussed in this section.

12.3.1 Matrix Inversion

The first example is adapted from [BFH 76]. Each iteration of a loop is supposed to read a matrix, invert it, and print the result. If the matrix is singular, a message should be printed and the program should proceed with the next matrix.

```
procedure P is
   procedure TREAT_MATRICES(N : INTEGER) is
      SINGULAR : exception;

      procedure INVERT(M : in out MATRIX) is
      begin
         -- compute determinant inverse
         -- note : this may implicitly raise DIVIDE_ERROR
         -- complete inversion of the matrix.
      exception
         when DIVIDE_ERROR => raise SINGULAR;
      end INVERT;

      procedure TREAT_ONE is
      begin
         READ(M);
         INVERT(M);
         PRINT(M);
      exception
         when SINGULAR => PRINT("Matrix is singular");
      end TREAT_ONE;

   begin  -- TREAT_MATRICES
      for I in 1 .. N loop
         PRINT("ITERATION");
         PRINT(I);
         TREAT_ONE;
      end loop;
   end TREAT_MATRICES;

begin  -- P
   TREAT_MATRICES(20);
end;
```

As this example illustrates, the occurrence of the predefined exception DIVIDE_ERROR within INVERT is expected and consequently an appropriate handler has been provided. On the other hand, an occurrence of this exception within READ or PRINT would cause termination of P, since no handler has been provided within P.

In order to illustrate the dynamic behavior of this program, let us consider the stack situation during a call to INVERT:

```
(1)  P                 calling TREAT_MATRICES
     TREAT_MATRICES    calling TREAT_ONE
     TREAT_ONE         calling INVERT
     INVERT            executing normal statements of INVERT
```

If DIVIDE_ERROR occurs during the inversions, the corresponding handler will be executed.

```
(2)  P                 calling TREAT_MATRICES
     TREAT_MATRICES    calling TREAT_ONE
     TREAT_ONE         calling INVERT
     INVERT            executing the handler for DIVIDE_ERROR
```

Note that during its execution, the handler has access to the local variables and parameters of INVERT. Here the only effect of this handler is to raise the exception SINGULAR. As a consequence, the activation of INVERT is deleted and the exception SINGULAR is propagated within TREAT_ONE at the point of call of INVERT. The handler of TREAT_ONE for SINGULAR is then executed.

```
(3)  P                 calling TREAT_MATRICES
     TREAT_MATRICES    calling TREAT_ONE
     TREAT_ONE         executing the handler for SINGULAR
```

In this case the execution of the handler terminates the execution of TREAT_ONE without propagating an exception in TREAT_MATRICES. This leads to the following stack configuration where another iteration of the loop can now be executed.

```
(4)  P                 calling TREAT_MATRICES
     TREAT_MATRICES    executing loop
```

The above example is characteristic of a family of problems in which a sequence of input objects must be subjected to a given treatment. Should this treatment fail for one object of the sequence, it would be unreasonable to abort the complete sequence. Rather, the exception handling facility provides the ability to do a partial termination, that of the current object.

12.3.2 *Division*

Consider the following definition of the function DIVISION:

```
function DIVISION (A, B : REAL) return REAL is
begin
   return A/B;
exception
   when DIVIDE_ERROR => return REAL'LAST;
end;
```

Should DIVIDE_ERROR occur during the computation of A/B, the execution of the handler will complete the execution of the function DIVISION. Any statement that is valid within the sequence of statements of DIVISION is also valid in the handler. Hence the handler may issue a return statement

```
return REAL'LAST;
```

on behalf of the function.

This example shows the nature of handlers. They must be viewed as substitutes ready to take charge of the operations in case of error.

12.3.3 File Example

This example illustrates a case where exception handling is used to treat an event which is certain to happen: reaching the end of a file. Naturally this example could be formulated with an explicit check for each iteration. Assuming the file to be quite large, however, the body of the procedure TRANSFER may be efficiently represented as an (apparently) infinite loop, and the final actions of the procedure performed by the exception handler END_OF_FILE.

```
procedure TRANSFER is
   use TEXT_IO;
   INF    : IN_FILE;
   OUTF   : OUT_FILE;
   C      : CHARACTER;

begin
   OPEN(INF, "SOURCE");
   OPEN(OUTF, "DESTINATION");
   loop
      GET (INF, C);
      PUT (OUTF, C);
   end loop;
exception
   when END_OF_FILE =>
      PUT(OUTF, EOF);
      CLOSE(INF);
      CLOSE(OUTF);
end;
```

The procedure TRANSFER transfers the characters from the file SOURCE into the file DESTINATION. At each iteration, GET is called and eventually an END_OF_FILE exception will occur. As a consequence the corresponding handler will be activated and its execution will complete the execution of TRANSFER.

This example shows that although many exceptions will correspond to error conditions, some of them may just be normal conditions for termination.

12.3.4 A Package Example

The example below reproduces a skeleton of the TABLE_MANAGER package described in the Reference Manual section 7.5.

```
package TABLE_MANAGER is
   type ITEM is ...
   procedure INSERT   (NEW_ITEM   : in   ITEM);
   procedure RETRIEVE (FIRST_ITEM : out  ITEM);
   TABLE_FULL : exception; -- may be raised by INSERT
end;

package body TABLE_MANAGER is
   ...
   procedure INSERT (NEW_ITEM : in ITEM) is
   begin
      if NO_MORE_SPACE then
          raise TABLE_FULL;
      end if;
      -- normal actions of INSERT
   end;
   ...
end TABLE_MANAGER;
```

The interface of the table manager defines the operations INSERT and RETRIEVE, and the exception TABLE_FULL. Any procedure such as Q that uses the package may provide a local handler for this exception:

```
procedure Q is
   use TABLE_MANAGER;
   ...
   procedure SAFE_INSERT(ELEMENT : in ITEM) is
      SPARE : ITEM;
   begin
      INSERT(ELEMENT);
   exception
      when TABLE_FULL =>
         RETRIEVE(SPARE);
         -- perform normal treatment of spare
         INSERT(ELEMENT);
   end SAFE_INSERT;
begin
   -- includes calls of SAFE_INSERT instead of INSERT
end Q;
```

Within procedure Q, a procedure SAFE_INSERT with a local handler for TABLE_FULL is provided. Should this exception be raised by the body of INSERT, the local handler for TABLE_FULL gains control and is able to perform a RETRIEVE before reissuing the same call to INSERT. Should an exception occur again in this second call, the execution of SAFE_INSERT will be terminated and the same exception will be raised in the calling environment.

It is worth mentioning that the body of INSERT is programmed in a *robust* manner: it keeps its environment intact if it cannot accomplish its tasks normally. It is this property that permits SAFE_INSERT to reissue a second call of INSERT when the first call fails.

12.3.5 *Example of Last Wishes*

The occurrence of an exception causes termination of the procedures in the dynamic chain of calls up to (and not including) the first procedure handling the exception. Suppose A handles a given exception, and

A calls B, B calls C, C calls D

If the exception occurs in D, the activations of D, C, and B will be terminated in that order.

One may want to let these procedures express their *last wishes* before disappearing, for instance, to perform some clean-up actions. This can be achieved by providing a handler for **others** in each of these procedures. The handler will then issue the statement

```
raise;
```

which reraises the same exception in the calling environment. We are thus able to achieve an effect similar to that of the UNWIND clause of Mesa [GMS 77] without introducing a special language construct. We illustrate last wishes with the example of a procedure performing operations on a file:

```
procedure OPERATE(F_NAME : STRING) is
   F : INOUT_FILE;
begin
   -- initial actions
   OPEN(F, F_NAME);
   -- perform work on the file
   CLOSE(F);
   -- final actions
end;
```

Should an exception occur while performing work on the file, it would be left in an open state when leaving the body of OPERATE. It is possible to avoid this by expressing the corresponding corrective action in a handler:

```
procedure SAFE_OPERATE(F_NAME : STRING) is
   F : INOUT_FILE;
begin
   -- initial actions
   OPEN(F, F_NAME);
   begin
      -- perform work on the file
   exception
      when others =>
         CLOSE(F);
         raise;
   end;
   CLOSE(F)
   -- final actions
end;
```

Now, if any exception occurs, either during the initial or the final actions it will be propagated to the caller of SAFE_OPERATE. If however, the exception occurs within the block, while performing work on the file, the inner handler will first close the file before propagating the exception.

Similar techniques can be used in parallel processing examples where a given task could be left in an inconsistent state waiting forever to receive the stop signal from a task that died after sending a start signal.

12.4 Exceptions and Parallel Processing

The exception handling facility has been presented so far mainly in terms of sequential programs. As a consequence the concepts presented are valid within a task.

The ability for one task to raise or to propagate an exception in another task must however be viewed as a possibility with potentially extremely severe consequences.

In no way should such external exceptions be considered as being normal terminating conditions. Interfering asynchronously with the execution of a task may catch it in a state where it is not prepared to respond to such intervention. There is then always a risk of leaving the task in a state of confusion and also, of contaminating other tasks that were communicating with it.

The normal means of communicating with a task is via entry calls. Hence most situations in which the termination of a task must be decided by another task should be programmed by calling a special entry, say TERMINATE, of the task to be terminated. The clear advantage of such a solution is the possibility thus offered to include accept statements for the TERMINATE entry at those places where the termination can be done in an orderly fashion.

For these reasons, the ability of a task to raise or to propagate an exception in another task has been substantially limited.

A first limitation of the exception handling facility for tasks concerns exception propagation. Contrary to what is done for subprogram and packages, if an exception is not serviced by a handler within a task, the exception is not further propagated; the task execution is merely terminated.

Note that if the exception were propagated to the parent task, it would mean that the child task could have an asynchronous interference on its parent. It would also mean that the latter could be subject to simultaneous such interferences from its local tasks.

The second limitation concerns the nature of the exceptions that can be raised explicitly by one task for another task. This ability is provided only for the predefined exception FAILURE.

Other exceptions may also arise during task initiation and also during communications between tasks. Unlike FAILURE, however, these are synchronous exceptions comparable to those that can be propagated by procedure calls. The predefined exception TASKING_ERROR corresponds to several such situations.

We next discuss the FAILURE exception and the situations in which a TASKING_ERROR may arise.

12.4.1 Raising the FAILURE Exception in Another Task

The only exception that can be raised explicitly by one task in another task is the predefined exception FAILURE. For a task T, the exception FAILURE is raised by a raise statement of the form

```
raise T.FAILURE;
```

The effect of this statement is as follows

(a) If the task T is currently being executed, its execution is interrupted.

(b) If the task T has issued an entry call, two cases may arise:

- If the entry call has not been accepted it is cancelled. The task owning the called entry is unaffected.
- If an accept statement for this entry is being executed, the exception TASKING_ERROR is raised within the unit containing this statement.

(c) The continuation address of the task is modified so as to reflect occurrence of the exception, and the task T is put in the ready queue of the scheduler.

(d) The execution of the task raising the exception may continue.

As for other exceptions a task may provide handlers for FAILURE. The task is thus given an opportunity to perform some clean up actions before terminating. The general schema of a task likely to be terminated by the FAILURE exception is as follows:

```
task body T is
   ...
begin
   -- normal actions of the task
exception
   when FAILURE =>
      -- perform clean-up actions, for example:
      -- raise FAILURE for local tasks
      -- issue some accept statements
end;
```

It should be clear that raising FAILURE for a task T cannot guarantee the termination of T since the handler can contain arbitrary statements. Hence this ability may be insufficient in situations where time is critical, and the abort statement appears as a necessary complement in such situations. The sequence:

```
raise  T.FAILURE;
delay  1.5*SECONDS;
abort  T;
```

can be used to give T the opportunity to perform clean-up operations and nevertheless make sure that its execution does not extend beyond a certain delay.

As an example of possible use of the FAILURE exception, consider a situation in which a task OPERATOR must be able to take some default actions in case it cannot obtain service from a SERVER within a certain delay. Instead of calling the entry REQUEST directly, OPERATOR will ask a local task AGENT to do so and to report completion.

```
task body OPERATOR is
   entry REPORT(I : INFO);

   task AGENT is
      entry ASK;
   end;

   task body AGENT is
      J : INFO;
   begin
      accept ASK;
      SERVER.REQUEST(J);
      OPERATOR.REPORT(J);
   exception
      when FAILURE => null;
   end;
begin
   initiate AGENT;
   AGENT.ASK;

   select
      accept REPORT(I : INFO) do
         -- the answer was received in time:
         -- perform normal operations
      end;
   or
      delay 10.0*SECONDS;
      -- no answer : perform default actions
      raise AGENT.FAILURE;
   end select;
end OPERATOR;
```

If the server does not answer the request within the delay, the task OPERATOR raises the exception FAILURE for the local task AGENT. Several situations are then possible:

(a) SERVER had not yet accepted the request (either because of malfunction or because it is still busy processing requests from other tasks), the request is just removed from the queue of the corresponding entry and SERVER is unaffected.

(b) SERVER is in the middle of the accept statement for the entry REQUEST, an exception TASKING_ERROR is raised within the accept statement.

(c) The request is completed after the delay. Then if AGENT has already issued a call to the entry REPORT, this call is cancelled.

This example illustrates the care that must be taken when raising the exception FAILURE. In general, it should be used only in extreme situations, for example, to protect a task against a possible malfunction in another task or to terminate an erroneous task.

Raising FAILURE for another task is a drastic measure that should only be used when normal means of communication have failed. Whenever termination can be planned, it is preferable to request it with an entry call.

Because of its importance, the FAILURE exception prevails over any other possible exception.

12.4.2 Exceptions Propagated During Communications Between Tasks

When two tasks are attempting to communicate with each other, or are engaged in a communication, an abnormal situation arising in one of them may have an effect on the other one.

As a basis for discussing the various cases that may arise, consider a task SERVER providing a procedure ASK and an entry UPDATE, and a task CALLER:

```
task SERVER is
   ...
   procedure ASK  (...);
   entry UPDATE(...);
end;

task body SERVER is
   ...
   procedure ASK  (...) is
      ...
   end ASK;
begin
   ...
   accept UPDATE (...) do
      -- statements for servicing the request
   end UPDATE;
   ...
end SERVER;

task body CALLER is
   ...
begin
   ...
   SERVER.ASK(...);
   ...
   SERVER.UPDATE(...);
   ...
end CALLER;
```

The task CALLER is calling the procedure ASK and the entry UPDATE. Abnormal situations will arise if SERVER is not active at the time of either call. Similarly an abnormal situation in SERVER may affect the caller.

12.4.3 Exceptions Propagated by Calls of Task Procedures

If an exception is raised within the body of the procedure ASK, and not serviced by a local handler, it is propagated to the caller, as usual.

Calling the procedure ASK is of course only possible if the task SERVER is active, otherwise, the exception TASKING_ERROR is propagated to the caller. In addition, there is the possibility that SERVER may become inactive (either by normal or by abnormal termination) during the execution of ASK. Again the caller will receive the TASKING_ERROR exception.

12.4.4 Inability to Achieve a Rendezvous

When CALLER issues a call to the entry UPDATE of SERVER, the latter task may be inactive, in which case the rendezvous cannot be achieved. It may also happen that SERVER is active at the time of the entry call but is not yet ready to accept it. The calling task must then be suspended until SERVER is ready. In the meantime, however, SERVER may become inactive (by normal or abnormal termination) and again, the rendezvous cannot be achieved.

In either of these situations, the predefined exception *TASKING_ERROR is raised* at the point of the entry call in the caller.

12.4.5 Abnormal Situations in an Accept Statement

Once a rendezvous is achieved, the accept statement is executed. During this execution three kinds of abnormal situations may arise:

(1) An exception is raised within the accept statement.

(2) A third task disrupts the called task (i.e. SERVER), for example, by an abort statement or by raising the FAILURE exception.

(3) A third task disrupts the calling task (i.e. CALLER), for example, by an abort statement or by raising the FAILURE exception.

We next analyze the consequences of each of these three possible abnormal situations both with respect to the task issuing the entry call and with respect to the task containing the accept statement.

An exception is raised within an accept statement

Consider a situation in which an exception, say *error*, is raised within the accept statement of SERVER:

```
task body CALLER is
   ...
   SERVER.UPDATE(...);
   ...
end USER;
```

```
task body SERVER is
   ...
   accept UPDATE(...) do
      ...
      error
      ...
   end UPDATE;
   ...
end SERVER;
```

From the point of view of the caller, the accept statement is analogous to a procedure body executed when the corresponding entry is called. Hence if an exception is raised (and not handled within the accept statement itself) it should be propagated at the point of call of SERVER.UPDATE.

However, from the point of view of the task containing the accept statement, this statement is a normal statement of its body. Hence if an exception is raised within SERVER, it should be handled by a handler provided within that same task.

To summarize, an ordinary exception (not FAILURE) raised within an accept statement and not handled there is propagated both in the calling and in the called tasks. Both tasks may provide handlers for the exception.

The called task is disrupted.

A different treatment must be employed if the called task (i.e. SERVER) receives a FAILURE exception raised by a third task. The main purpose of this exception is to provide a possibility of clean termination for a task. Hence it would be undesirable to propagate this special exception to a calling task. For this reason, the calling task will not receive FAILURE, but rather, the less severe exception TASKING_ERROR.

Similarly TASKING_ERROR is raised in the caller if the called task is aborted by a third process while executing the accept statement.

The calling task is disrupted.

In this case there is no point in pursuing the execution of the accept statement since the caller is no longer waiting for its completion. Hence, the execution of the accept statement is abnormally terminated and the exception TASKING_ERROR is raised in the place of this statement. Note that this case may result in disrupting another task as in case 2 or 3 above.

It is quite important to realize that this exception is not raised *within* the accept statement itself but rather outside it, for two reasons.

First, if the exception were raised inside the accept statement, it would be possible to handle it locally and thus to continue the communication with a possibly inactive task.

The second reason concerns the possibility of nesting accept statements. In such a case, should the callers for both the outer and the inner accept statements be aborted, the called task may receive two simultaneous TASKING_ERROR exceptions. However, no confusion is possible since the termination of the outer accept statement guarantees that of the inner accept statement.

12.4.6 Example of Propagation of Normal Exceptions for Tasks

The following program fragment shows a task USER which is starting a file transfer performed asynchronously by a task SPOOLER. The procedure OPEN_IN_OUT is used by SPOOLER to open the two files. If either of the files is invalid, the corresponding exception is raised after possibly closing the other file.

Two forms of input output exceptions may be raised within the body of the task SPOOLER. The exception END_OF_FILE is handled locally and not propagated. The exception FILE_NAME_ERROR may be raised within the accept statement for the entry START_TRANSFER. The handler provided within SPOOLER simply prepares it for another iteration.

```
task SPOOLER is
   entry START_TRANSFER(SOURCE, DESTINATION : in FILE_NAME);
end;

task body SPOOLER is
   INF     : IN_FILE;
   OUTF    : OUT_FILE;
   C       : CHARACTER

   procedure OPEN_IN_OUT (SOURCE, DESTINATION : in FILE_NAME) is
   begin
      OPEN(INF, SOURCE);
      begin
         OPEN(OUTF, DESTINATION);
      exception
         when FILE_NAME_ERROR => CLOSE(INF); raise;
      end;
   end;

begin
   loop
      begin
         accept START_TRANSFER(SOURCE, DESTINATION : in FILE_NAME) do
            OPEN_IN_OUT(SOURCE, DESTINATION);
         end;

         loop
            GET (INF,     C);
            PUT (OUTF,    C);
         end loop;
      exception
         when END_OF_FILE =>
            PUT(OUTF, EOF);
            CLOSE(INF);
            CLOSE(OUTF);
         when FILE_NAME_ERROR => null; -- restart main loop
      end;
   end loop;
end SPOOLER;
```

In addition, the occurrence of the exception FILE_NAME_ERROR within the accept statement for START_TRANSFER also has an effect on the calling task USER. The exception is propagated in that task where it can be serviced by a local handler:

```
task body USER is
   I_F, O_F : FILE_NAME;
begin
   ...
   begin
      SPOOLER.START_TRANSFER(I_F, O_F);
   exception
      when FILE_NAME_ERROR =>
         -- do something on I_F and O_F
   end;
   ...
end;
```

12.4.7 Example of TASKING ERROR in Nested Accept Statements

Nested accept statements are illustrated by the example of a task FILTER buffering characters received through the entry WRITE from slow producers. These characters can be read in batches by other tasks through the entry READ.

When a call to READ occurs, two cases are possible: either FILTER has buffered enough characters to forward them immediately, or it has not; in the latter case the reader must wait until enough characters have been received. This second possibility is handled by nesting an accept statement for WRITE within the one for READ.

Abnormal termination of either a reader or writer should not affect FILTER. In particular, if a writer dies while calling WRITE, this should not terminate the READ rendezvous. This isolation is provided by embedding the accept statement for WRITE within a block that provides a null handler for TASKING_ERROR.

Note also that the incrementation of the indexes is done first within temporary variables. These are updated outside of the accept statements. Thus the values of the indexes are maintained as correct even in the case of exceptions.

```
task FILTER is
   LENGTH : constant INTEGER := 120;
   entry READ (BATCH : out STRING(1 .. LENGTH));
   entry WRITE(C : CHARACTER);
end;
```

```
task body FILTER is

   MAX_BUF  : constant INTEGER := 10*LENGTH;
   BUFFER   : array (1 .. MAX_BUF) of CHARACTER;
   COUNT    : INTEGER range 0 .. MAX_BUF := 0;
   IN_INDEX, OUT_INDEX : INTEGER range 1 .. MAX_BUF := MAX_BUF;
   TEMP_IN  : INTEGER range 1 .. MAX_BUF := IN_INDEX;
   CHAR     : CHARACTER;

   procedure INPUT_CHAR is
   begin
      IN_INDEX := (IN_INDEX mod MAX_BUF) + 1;
      BUFFER(IN_INDEX) := CHAR;
      COUNT := COUNT + 1;
   end;
begin
   loop
      begin
         select
            when COUNT < MAX_BUF =>
               accept WRITE (C : CHARACTER) do
                  CHAR := C;
               end;
               INPUT_CHAR;
         or
            accept READ (BATCH : out STRING(1 .. LENGTH)) do
               while COUNT < LENGTH loop
                  begin
                     accept WRITE (C : CHARACTER) do
                        CHAR := C;
                     end;
                     INPUT_CHAR;
                     TEMP_IN := IN_INDEX;
                  exception
                     when TASKING_ERROR => null;  -- isolates from writer termination
                  end;
               end loop;
               for I in 1 .. LENGTH loop
                  BATCH(I) := BUFFER((OUT_INDEX+I) mod MAX_BUF + 1);
               end loop;
            end READ;
            OUT_INDEX := (OUT_INDEX+LENGTH-1) mod MAX_BUF + 1;
            COUNT := COUNT - LENGTH;
         end select;
      exception
         when TASKING_ERROR =>
            IN_INDEX := TEMP_IN;
            if IN_INDEX >= OUT_INDEX then
               COUNT := IN_INDEX - OUT_INDEX;
            else
               COUNT := IN_INDEX + MAX_BUF - OUT_INDEX;
            end if;
      end;
   end loop;
end FILTER;
```

12.5 Technical Issues

The discussion of exception handling in [Go 75] classifies exceptions into three categories.

(a) *Escape* exceptions which require termination of the operation raising the exception.

(b) *Notify* exceptions, which forbid termination of the operation raising the exception and require its resumption after completion of the handlers' actions.

(c) *Signal* exceptions, which leave the choice to the handler between termination and resumption.

The Steelman requirements specify that the occurrence of an exception should cause transfer to a handler without completing the elaboration of the declaration or the execution of the statement where the exception occurred. Hence they rule out *notify* and *signal* exceptions, and quite justifiably so, since these forms of exceptions violate program modularity and make optimization difficult if not impossible.

Exceptions in the Green language are thus of the *escape* form; they serve only for error situations and as terminating conditions, permitting a simpler language formulation.

The technical problems concerning the interactions between exceptions and parallelism have been mentioned in the previous section. The key idea was to provide a simple rule for cases where simultaneous exceptions occur in a given task. For the TASKING_ERROR exception, multiplicity can only occur in case of nested accept statements, and the outer exception prevails. The exception FAILURE prevails over other exceptions and simultaneous occurrences of this exception are treated as a single occurrence.

The remainder of this discussion will concentrate on the following issues:

- Exceptions during the elaboration of declarations
- Propagation of exceptions beyond their scope
- Suppression of exceptions
- Implementation of exception handling
- Proving programs with exceptions

12.5.1 Exceptions During the Elaboration of Declarations

The elaboration of declarations may involve the evaluation of some expressions and in consequence, exceptions may be raised during this elaboration. Consider for example the procedure

```
procedure A(N : INTEGER) is
   C  : constant INTEGER := N*N;
   D  : INTEGER := C;
   T  : array (1 .. C) of INTEGER;
begin
   -- statements of A
exception
   -- handlers of A
end;
```

If an exception occurs during the elaboration of the constant C, the procedure will be in a state where D is not initialized, and the space for the array T is not yet allocated. Consequently, a handler may not be able to do much; any reference to D or T will be erroneous and may cause a further exception.

When this risk is important, it is always possible to replace the expressions that may cause exceptions by function calls and to provide handlers within these functions. Another possibility is to provide a special boolean flag:

```
procedure A(N : INTEGER) is
   ELABORATION_DONE :   BOOLEAN := FALSE;
   ...
begin
   ELABORATION_DONE :=  TRUE;
   ...
exception
   -- handlers may test if ELABORATION_DONE
end;
```

The design also considered an alternate semantics in which the handlers are only applicable after completion of the elaboration of declaration (hence, an exception arising during the elaboration of the local declarations of a procedure would always be propagated). This alternate semantics was rejected for implementability reasons.

12.5.2 Propagation of Exceptions Beyond Their Scope

Since exceptions can be propagated they can be propagated beyond their scope. It is even possible for an exception to be propagated outside its scope and back within its scope. Thus, in the following example, if B calls OUTSIDE and OUTSIDE calls A, the exception ERROR raised within A will be propagated to OUTSIDE and again to B:

```
package D is
   procedure A;
   procedure B;
end;

procedure OUTSIDE is
begin
   ...
   D.A;
   ...
end;

package body D is
   ERROR : exception;
   procedure A is
   begin
      ... raise ERROR; ...
   end;

   procedure B is
   begin
      ...
      OUTSIDE;
      ...
   exception
      when ERROR =>
         -- ERROR may be propagated by OUTSIDE calling A
   end;
end D;
```

In general, an exception propagated beyond its scope can only be handled by a handler for **others**. It can be further propagated or reraised by the anonymous form of the raise statement (**raise**;).

This rule provides a simple and consistent interpretation of the above example and it avoids the complexity and runtime costs that would be incurred if exceptions propagated beyond their scope were converted into a unique undefined exception. This design also considered, and rejected, the possibility of associating the names of the possibly propagated exceptions with each procedure declaration. The main reason for rejecting this possibility is the fact that this would require extra runtime code for filtering the propagation of exception. For example, if a procedure were declared as

```
procedure P(X : INTEGER) propagates A, B, C;  -- not legal in Green
```

its body would have to be compiled as the equivalent of the following procedure:

```
procedure P(X : INTEGER) is
   ...
begin
   ...
exception
   when A | B | C => raise;
   when others => raise anonymous_exception;
end P;
```

We considered the resulting code expansion to be too prohibitive, especially in the case of small functions and procedures.

With the solution adopted in Green, the user can always put similar information in comment form. The case **others** covers all possible anonymous exceptions, not just one.

The internal codes associated with exceptions must all be distinct - as are the codes of the different literals of an enumeration type. In general, this assignment of codes to exceptions must be performed at linkage editing time.

12.5.3 Suppression of Exceptions

A given program unit, and in particular a procedure, may be *robust*, in that it will perform some computation, and produce some result for any value of its input parameters. On the other hand, its validity may be guaranteed for only certain values of these parameters. The exception mechanism is a useful tool to achieve robustness, but this may be gained at some cost in efficiency, since detection of some error situations may be expensive, unless aided by special hardware.

In some cases where robustness is not an issue, or can be attained by means other than runtime checks, the programmer may not wish to incur the cost of checking for possible exception situations. The pragma

pragma SUPPRESS (*list of exception names*);

indicates that no check need be performed for the designated exception situations. (Should such a situation occur, behavior of the program would be unpredictable.)

As a consequence, the compiler may suppress checks for such situations, and will do so if it results in an optimization. Note that this pragma is an indication that the user does not care, as far as the validity of the program is concerned, whether the checks are performed or not.

However, in the case of exceptions whose detection is aided by special hardware, inhibiting the corresponding hardware mechanisms may be costlier than actually performing the checks, hence the pragma is not imperative. It does not mean that the checks are not done.

An alternative view of the SUPPRESS pragma would be that of a directive indicating imperatively that no check is to be performed to detect the exception. This approach would correspond to a decision to continue execution of a program *in spite of* any erroneous situation. It would give an appearance of robustness which might be exploited in cases where the programmer knows that the abnormal situation will have some effects that can be detected at a later time, but it is in opposition to the general philosophy of the language.

In addition, the need to provide a semantics that reconciles software and hardware detected exceptions would have a negative effect on the efficiency of the programs. If the pragma were imperative, then on a machine with hardware detected exceptions, it would be necessary to inhibit the hardware checks for a scope in which the corresponding exception has been suppressed. Thereafter, it would be necessary to reenable the hardware detection prior to each call to a unit outside that scope, and again to inhibit the detection following a subsequent return from the call.

12.5.4 Implementation of Exception Handling

One important design consideration for the exception handling facility is that exceptions should add to execution time only if they are raised.

Several techniques may be used to reach that goal, and they may differ from one implementation to another. The essential idea is that all processing costs should be concentrated on the treatment of the raise statement. As a consequence, processing of a raise statement may be relatively slow. In contrast, the elaboration of an exception declaration or that of the body of an exception handler represents only translation time and space costs. They have no influence on the execution time.

As a feasibility proof, we outline a possible implementation technique that no runtime costs whatsoever are incurred for exceptions unless they are raised. This technique has been used for some debugging systems and it bears some resemblance to the technique used in Mesa [GMS 77]. The basic principles are as follows:

(a) When an exception occurs, the specific runtime system that treats exceptions must be able to locate the addresses of the currently active procedure calls. This condition is satisfied if, as is usually the case, return addresses are stored in procedure activations.

(b) Knowing the address of the code of a procedure, it must be possible to locate the address of the code of the first handler. Similarly, given a handler, it must be possible to find the next handler. This condition can be satisfied by chaining the handlers and by storing the address of the first handler just before the code of the procedure.

(c) Each handler must start with the indication of the exception code (or codes) that it services. Some convention must be used for the handler for **others** which must appear last.

(d) When performing the association of an exception with a handler, the runtime system locates the procedure address and from there the chain of handlers. It may then inspect the exception codes to find the appropriate handler, if any.

We reiterate that this solution should only be considered as an existence proof that exceptions may be implemented at no cost unless raised. Other techniques may be more suitable on alternative machines.

12.5.5 Proving Programs with Exceptions

The problems of exception handling facilities such as the PL/I *on conditions*, which permit resumption after exception, are well known. For instance, consider the consecutive statements.

```
X  := P + Q;
Y  := X - Q;
assert (Y = P);
```

Unless overflow occurs in the evaluation of P + Q, the final assertion should be satisfied. This however would not be true if a handler for overflow were able to provide a different value for X and return to the same statement list.

This simple example shows the near impossibility of proving programs with unconstrained *signal* and *notify* exceptions. For the same reasons, such programs are extremely difficult to optimize.

In contrast, for the proposal given in the Green language, simple proof rules may be given as has been shown by Bron [BFH 76] and Fokkinga [Fo 77]. Additional examples may be found in the reference [DH 76]. The main idea is a consequence of the definition of the role of a handler.

As mentioned above, when an exception occurs in a procedure, the execution of the handler replaces the remainder of the execution of the procedure considered. Consequently the effect of a procedure is achieved.

(a) either by its body if no exception occurs

(b) or by the part of its body until the point where the exception E occurs and then by the handler for E if E occurs.

Two simple cases have been shown in the programming examples:

(1) In the SAFE_INSERT example of section 12.3.4, we have shown a case where these two rules reduce to a simpler form. The effect of SAFE_INSERT is achieved.

    (a) either by its body if no exception occurs

    (b) or by the handler TABLE_FULL if TABLE_FULL occurs

(2) In the file example of section 12.3.3 where the exception END_OF_FILE is used as a terminating condition, the effects of the procedure TRANSFER is achieved by the succession of the effects of

    (a) its body

    (b) the body of the handler END_OF_FILE

This shows that with adequate programming conventions, the effect of a procedure containing an exception handler can be characterized in a simple way. This simplifies correctness proofs. On the other hand, proving a program where exceptions arise in parallel tasks remains a considerably more difficult problem.

13. Generic Program Units

13.1 Introduction

Generic clauses provide a general facility for translation time parameterization of program units. As with other parameterization mechanisms, the primary purpose of this generic facility is factorization, so as to bring about a reduction in the size of the program text while simultaneously improving both readability and efficiency.

Generic clauses may be viewed as a natural extension of subprogram parameterization. When otherwise identical actions differ by the particular value or variable, these actions may be incorporated in a subprogram where that value or variable appears as a parameter. Having thereby *factorized* the common parts, the text becomes smaller and easier to read and *clerical* errors, involving accidental lack of identity among the several copies, are eliminated. Moreover, the translator can take advantage of this commonality to produce more compact code.

Traditional parameterization mechanisms are usually restricted to variables, although the same factorization arguments can apply when two otherwise identical program units differ by some other property, such as a type.

A classical example is provided by stacks. In the Green language, stacks would be typically formulated as a data type, encapsulated with its associated operations within a package. Although one may want to have stacks of integers and stacks of real numbers, it is clear that neither the stack algorithms (nor the proof of their correctness) depend upon the type of the items to be stacked.

Strong typing prevents the writing of a single procedure to deal with objects which may be either integers, reals or any subsequently defined type. Hence another language mechanism is needed to achieve this kind of parameterization by a type; this mechanism is the generic clause.

A generic clause permits parameterization of the text of a package or of other program units. Replication of text can thereby be avoided, yielding better readability. In addition the translator may use its knowledge of data type representations to achieve certain optimizations (e.g. reusing the same code for stacks of integers and reals if these types occupy the same number of bits). Seen in this light, such a generic facility provides a natural complement to strong typing, minimizing the unnecessary duplication of both text and code.

One of the most common applications of any generic facility is factoring out dependencies on particular data types. Several existing languages have accordingly introduced language features to accommodate this sort of parameterization. By far the most powerful is that provided by the language EL 1 [We 74]; this generality is achieved, however, at the cost of interpreting types in a fully dynamic fashion, which is incompatible with the efficiency and security criteria imposed in the present context.

Languages such as Simula [DNM 69], Clu [Li 74] and Mary [R 74] offer a reasonably elegant approach to this problem, but all impose the constraint that all objects are handled by reference. This introduces additional overhead (i.e., indirect access) even in cases where such generality is neither needed nor wanted. The language Euclid [LHLMP 76] provides a facility which is only applicable when the alternative types are specified in advance and can therefore appear as variants of some parameterized type; the language CS 4 [Br 75] makes available a limited facility that may only be used in conjunction with predefined types. Neither approach offers the flexibility that is required when the definition of new data types is viewed as the rule rather than the exception.

A review of the shortcomings of existing mechanisms which allow types to be used as parameters showed that it was inappropriate to introduce additional language features solely for this purpose, principally because of their relative complexity. This conclusion is substantiated by noting that essentially the same effect (and many others as well) can be achieved by far simpler means using traditional macro-expansion techniques (although in a context sensitive manner). The problem then reduces to integrating this well-established approach into the framework of a high order language at reasonable cost.

In the Green language, more sophisticated sorts of parameterization are accommodated by generic program units, which are a restricted form of context sensitive macro facility. The main objectives in providing this particular mechanism have been:

- to allow an additional degree of freedom in factorization without sacrificing efficiency;
- to permit the translator to take advantage of this factorization to minimize the size of the code;
- to preserve the security that is present for ordinary, unparameterized program units;
- to introduce only a modest extension, with relatively small impact on the rest of the language and its translator.

13.2 Informal Presentation of Generic Facilities

Any program unit (that is, any subprogram or module) may be declared with a generic clause. This clause defines translation-time parameters which may appear in the body of the program unit.

A generic program unit is not a normal program unit. A generic subprogram cannot be called; it is rather a *model* or *template* for all program units which can be obtained by providing translation-time substitutions for the generic parameters.

A specific program unit which corresponds to a given model is created by a declaration called a *generic instantiation*. This has the effect of creating a named instance. In the case of a subprogram, for example, this named instance can then be called in the usual way. Thus, apart from parameterization, generic declaration is for program units what a type declaration is for data objects:

	data object	*program units*
defining the model:	type declaration	generic declaration
defining an instance:	object declaration	generic instantiation

13.2.1 Generic clauses

A generic clause can be given as a prefix in a subprogram or module specification. It has the form

generic [(generic_parameter {; generic_parameter})]

where a generic parameter is a normal parameter declaration, a subprogram specification, or a type specification of the form

[**restricted**] **type** identifier

As an example, consider the following generic procedure:

```
generic(type ELEM)
procedure EXCHANGE(X, Y : in out ELEM) is
   TEMP : ELEM;
begin
   TEMP := Y;
   Y    := X;
   X    := TEMP;
end;
```

In this example X and Y are the normal parameters of the procedure, which are subject to dynamic replacement. In contrast, the type ELEM given in the generic clause is a generic parameter which is to be substituted at translation time. This generic parameter may appear in the body of the generic subprogram; here it is used in the declaration of the variable TEMP.

13.2.2 Generic Instantiation

A generic instantiation creates an actual instance of the specified program unit by replacement of the generic parameters. Such an instantiation must appear in a declaration which gives the name of the particular instantiation. Generally, there will be several different instantiations of a given generic unit.

```
procedure SWAP_INT     is new EXCHANGE(INTEGER);
procedure SWAP_CHAR    is new EXCHANGE(CHARACTER);
procedure SWAP_COLOR   is new EXCHANGE(COLOR);
```

The resultant program units are ordinary procedures, applicable to arguments of the corresponding type.

```
SWAP_INT(I, J);
SWAP_COLOR(SHADE, TINT);
```

The fact that these procedures are obtained by generic instantiation does not preclude overloading of the names of operations.

```
procedure SWAP is new EXCHANGE(INTEGER);
procedure SWAP is new EXCHANGE(CHARACTER);
procedure SWAP is new EXCHANGE(COLOR);
```

In general, a generic instantiation has the form

```
new name [(generic_association {, generic_association})]
```

It can appear in subprogram declarations of the form

```
subprogram_nature designator is generic_instantiation;
```

Note that this form of subprogram declaration does not contain parameter declarations since the parameters of the subprogram thus declared are derived from those of the corresponding generic subprogram. Similarly, a generic instantiation can appear in a module declaration of the form:

```
module_nature identifier [(discrete_range)] is generic_instantiation;
```

The parameter associations given in generic instantiations have the same form as those given for subprogram calls, but additional forms exist for generic parameters that are types or subprograms since these are not possible as subprogram parameters. Thus the form

```
[formal_parameter is] type_mark
```

is used for parameters that are types, and the form

```
[formal_parameter is] [name.]designator
```

is used for parameters that are subprograms. Note that both named associations and positional associations are possible as usual. Thus our previous example can be equivalently written as:

```
procedure SWAP is new EXCHANGE(ELEM is INTEGER);
procedure SWAP is new EXCHANGE(ELEM is CHARACTER);
procedure SWAP is new EXCHANGE(ELEM is COLOR);
```

The resultant program unit, obtained by generic instantiation, can be viewed as a copy of the corresponding generic unit where each formal parameter has been replaced by the corresponding actual parameter. For example, the declaration of SWAP_INT produces a procedure equivalent to

```
procedure SWAP_INT(X, Y : in out INTEGER) is
   TEMP : INTEGER;
begin
   TEMP := Y;
   Y    := X;
   X    := TEMP;
end;
```

A generic instantiation need not be performed in the same declarative part as the corresponding generic declaration; it may be performed at any point where the name of the generic unit is visible.

The rule followed for the identification of names within a generic unit in such a case is similar to that used for subprograms. All non-local identifiers of the body of a generic unit (other than the generic parameters) are identified in the context of the generic *declaration.* In contrast, the actual parameters given in the generic associations must be interpreted in the context of the generic *instantiation.*

Note that this rule differs from a simple textual substitution. In the latter case all identifiers, including non-local ones, would be interpreted in the context of the instantiation. Hence it would not be possible in general to obtain the effect of generic program units by a simple (context free) macro facility.

To summarize, the generic parameter names are the only unresolved identifiers in the body of a generic program unit. For any generic instantiation, replacements must be provided for all generic parameters. These replacements are to be interpreted in the context of the instantiation.

13.2.3 Types as Generic Parameters

In the simple SWAP example presented so far, very little information about the type given as a generic parameter is needed. The only operation assumed available for objects of the type is assignment. Hence the procedure can be applied to any type for which assignment is defined: all types except restricted private types.

In general, when a type is specified as a generic parameter, it must be considered as a private type, with no available operations, aside from assignment and the predefined comparison for equality or inequality. Even these operations are denied if the generic type is a restricted type specified as

```
restricted type T
```

Operations on objects of such types (whether restricted or not) that are used within the generic body must be given by other generic parameters. As an example, consider the generic function

```
generic( type ELEM;
         function "*" (U, V : ELEM) return ELEM)
function SQUARING(X : ELEM) return ELEM is
begin
   return X * X;
end;
```

Since nothing is known a priori about the type ELEM it would not be possible to write X * X if the specification of "*" were not provided explicitly as a parameter of the generic clause. Instances of SQUARING can be created by supplying the corresponding parameters. For example, for

```
function SQUARE is new SQUARING(INTEGER, "*");
```

the operation "*" is the operation defined as

```
function "*" (U, V : INTEGER) return INTEGER;
```

hence, the normal integer multiplication. As a consequence, the generic instantiation produces a function body equivalent to the following:

```
function SQUARE(X : INTEGER) return INTEGER is
begin
   return X * X;
end;
```

Of course, other instantiations are possible. For example, we may want to use SQUARING for vectors, to extend the component by component multiplication

```
function MULT(X, Y : VECTOR) return VECTOR;
```

Thus with the generic instantiation

```
function SQUARE is new SQUARING(VECTOR, MULT);
```

we obtain a function returning the component by component square of a vector.

Default values can be defined for generic parameters that are subprograms. Thus an alternate form of definition for SQUARING is

```
generic ( type ELEM;
          function "*" (U, V : ELEM) return ELEM is ELEM."*")
function SQUARING(X : ELEM) return ELEM is
begin
   return X * X;
end;
```

The declaration of the functional parameter specifies that ELEM."*", the operation "*" defined for ELEM, must be used in the absence of an explicit parameter. Thus, since "*" is defined for INTEGER, the declaration of SQUARE can be written as

```
function SQUARE is new SQUARING(INTEGER);
```

Similarly the short form can be used for VECTOR if a user defined "*" operation exists for vectors in the scope of the instantiation. For other examples of such default parameters, the reader can see their use for the formulation of the standard input-output package in Chapter 14 of the Reference Manual.

To summarize, any operation applicable to a type specified as a generic parameter must also be an explicit parameter of the generic clause. Hence every use of such an operation in a generic unit can be identified, and the translator can check the correctness of its text independently from the instantiations, as for program units that are not generic. The information contained in the generic clause guarantees that any instantiation that matches the generic clause does produce a correct text.

13.3 The Use of Generic Program Units

This section contains a number of examples illustrating the use of generic program units.

13.3.1 Examples of Generic Functions

The following program fragment defines a generic function POWER to raise the value of an object of a type T to its *n*th power. This exponentiation is defined by repeated multiplication and the corresponding multiplication operation must be supplied as an actual generic parameter.

```
subtype NATURAL is INTEGER range 1 .. INTEGER'LAST;

generic( type T;
         function OPER(X, Y : T) return T)
function POWER(A : T; N : NATURAL) return T is
   RESULT : T := A;
begin
   for I in 2 .. N loop
      RESULT := OPER(RESULT, A);
   end loop;
   return RESULT;
end POWER;
```

This generic function can be used to define exponentiation for types for which a multiplication operation is known. For example:

```
function "**" is new POWER(RATIONAL, "*");
function "**" is new POWER(VECTOR, MULT);
```

Each of these declarations defines an overloading of the operator **, obtained by generic instantiation. For example, the first declaration defines a function whose specification is equivalent to

```
function "**" (A : RATIONAL; N : NATURAL) return RATIONAL;
```

It can be used to exponentiate rational numbers by repeated application of the multiplication operation defined for this type. Note also that the generic function can be used to apply any meaningful operation repeatedly, for example multiplication of a rational by an integer performed by repeated additions.

```
function "*" is new POWER(RATIONAL, "+");
```

We next consider a variation of the previous generic function in which a unary function is applied repeatedly:

```
generic( type T;
         function NEXT(X : T) return T)
function INVOLUTION(A : T; N : NATURAL) return T is
   RESULT : T := A;
begin
   for I in 1 .. N loop
      RESULT := NEXT(RESULT);
   end loop;
   return RESULT;
end INVOLUTION;
```

This generic function can be used to apply any unary function repeatedly, for example, to produce the *n*th successor or predecessor of an enumeration value

```
function SUCC  is new INVOLUTION(COLOR, COLOR'SUCC);
function PRED  is new INVOLUTION(COLOR, COLOR'PRED);
```

Again, these declarations produce functions whose specifications are equivalent to

```
function SUCC (A : COLOR; N : NATURAL) return COLOR;
function PRED (A : COLOR; N : NATURAL) return COLOR;
```

Similar functions can be instantiated to find the *n*th successor or predecessor of an item in a list:

```
function SUCC (X : LIST_ITEM) return LIST_ITEM is
begin
   return X.SUCC;
end;

function PRED(X : LIST_ITEM) return LIST_ITEM is
begin
   return X.PRED;
end;

function SUCC is new INVOLUTION(LIST_ITEM, SUCC);
function PRED is new INVOLUTION(LIST_ITEM, PRED);
```

Note that these involutions overload (but do not hide) the functions SUCC and PRED. Actually, the immediate successor of an element can be obtained in three ways:

```
X.SUCC              -- using the component SUCC
SUCC(X)             -- the unary function
SUCC(X,1)           -- the involution
```

13.3.2 An Example of a Generic Package

A discussion of generics would probably not be complete without a presentation of the treatment of either stacks or queues. Since the example of stacks has already been given in the Green Reference Manual, we shall give here a formulation of queues.

The specification of the generic package contains two generic parameters : ITEM, the type of the items in the queue, and SIZE, the size of each queue.

```
generic(type ITEM; SIZE : NATURAL)
package ANY_QUEUE is
   restricted type QUEUE is private;

   function    EMPTY  (Q : in QUEUE) return BOOLEAN;
   procedure   ADD    (X : in ITEM; Q : in out QUEUE);
   procedure   REMOVE(Q : in out QUEUE);
   function    FRONT  (Q : in QUEUE) return ITEM;

   QUEUE_OVERFLOW, QUEUE_UNDERFLOW : exception;
private
   type QUEUE is
      record
         STORE       : array (1 .. SIZE) of ITEM;
         COUNT       : INTEGER range 0  .. SIZE := 0;
         IN_INDEX    : INTEGER range 1  .. SIZE := 1;
         OUT_INDEX   : INTEGER range 1 .. SIZE := 1;
      end record;
end;
```

The package body provides the bodies of the functions and procedures promised in the specification.

```
package body ANY_QUEUE is

   function EMPTY(Q : in QUEUE) return BOOLEAN is
   begin
      return Q.COUNT = 0;
   end EMPTY;

   procedure ADD(X : in ITEM; Q: in out QUEUE) is
   begin
      if Q.COUNT < SIZE then
         Q.STORE(Q.IN_INDEX) := X;
         Q.IN_INDEX := (Q.IN_INDEX mod SIZE) + 1;
         Q.COUNT := Q.COUNT +1;
      else
         raise QUEUE_OVERFLOW;
      end if;
   end ADD;

   procedure REMOVE(Q : in out QUEUE) is
   begin
      if Q.COUNT > 0 then
         Q.OUT_INDEX := (Q.OUT_INDEX mod SIZE) + 1;
         Q.COUNT := Q.COUNT - 1;
      else
         raise QUEUE_UNDERFLOW;
      end if;
   end REMOVE;

   function FRONT(Q : in QUEUE) return ITEM is
   begin
      if Q.COUNT > 0 then
         return Q.STORE(Q.OUT_INDEX);
      else
         raise QUEUE_UNDERFLOW;
      end if;
   end FRONT;

end ANY_QUEUE;
```

Having defined ANY_QUEUE, it is now possible to instantiate two packages dealing respectively with queues of integers and queues of reals:

```
package ANY_INT_QUEUE    is new ANY_QUEUE(ELEM is INTEGER, SIZE := 100);
package ANY_REAL_QUEUE   is new ANY_QUEUE(ELEM is REAL,    SIZE := 100);
```

Semantically, these two declarations have created two packages which may be used as ordinary packages. In the present case, given that the same value has been given for SIZE, the translator may be able to reuse the same code for the procedures of the two modules, if reals and integers occupy the same number of bits.

A block dealing with real queues may appear as below:

```
declare
   use ANY_REAL_QUEUE;
   QA, QB : QUEUE;
begin
   ADD(3.14, QA);
   ...
   if FRONT(QA) = FRONT(QB) then
      REMOVE(QA);
      ADD(FRONT(QB) + 1.0, QA);
   end if;
   ...
end;
```

With the use clause for ANY_REAL_QUEUE, the type QUEUE is visible and can be used to declare the queues of reals QA and QB.

A slight difficulty exists if one wants to use both ANY_REAL_QUEUE and ANY_INT_QUEUE in the same scope, since both declare the type QUEUE. The naming conflict is resolved by naming the type as a selected component:

```
declare
   use ANY_REAL_QUEUE, ANY_INT_QUEUE;
   QA  : ANY_REAL_QUEUE.QUEUE;
   QJ  : ANY_INT_QUEUE.QUEUE;
begin
   ...
   ADD(3.0E5, QA);
   REMOVE(QJ);
   ...
   ADD(15, QJ);
   ...
end;
```

Generally, using selected components for the names of types will be sufficient (their repeated use can be avoided by declaring corresponding subtypes). Thereafter functions and procedures (such as ADD) appear as overloaded subprograms and no confusion is possible. For example the expanded specifications of ADD are equivalent to

```
procedure ADD(X : in REAL;    Q : in out ANY_REAL_QUEUE.QUEUE);
procedure ADD(X : in INTEGER; Q : in out ANY_INT_QUEUE.QUEUE);
```

In the case of the exceptions QUEUE_OVERFLOW and QUEUE_UNDERFLOW overloading is of no help and either selected components or renaming declarations must be used.

A final word on these two exceptions: the bodies of ADD, REMOVE and FRONT have been written so that no damage occurs to the queue if either exception occurs. As a consequence it is possible to provide a local handler for these exceptions:

```
declare
   use ANY_REAL_QUEUE, ANY_INT_QUEUE;
   subtype INT_QUEUE is ANY_INT_QUEUE.QUEUE;  -- INT_QUEUE defined as an abbreviation
   INTQ_ERROR : exception renames ANY_INT_QUEUE.QUEUE_OVERFLOW;
   QJ : INT_QUEUE;
   ...
begin
   ...
   ADD(3, QJ);
   ...
exception
   when INTQ_ERROR =>
      -- actions that will terminate the execution of the block in case QJ overflows.
end;
```

13.3.3 An Example of a Generic Task

As a further example consider the buffering interposed between a producer and a consumer (see the example of section 9.12 of the reference manual). This might be reformulated as a generic task where the type of the buffered items as well as the size of the buffer in question have been factored out as generic parameters.

```
generic(type ITEM; SIZE : INTEGER := 100)
task BUFFERING is
   entry READ (C : out  ITEM);
   entry WRITE(C : in   ITEM);
end;

task body BUFFERING is
   POOL    : array(1 .. SIZE) of ITEM;
   COUNT   : INTEGER range 0 .. SIZE := 0;
   IN_INDEX, OUT_INDEX : INTEGER range 1 .. SIZE := 1;
begin
   loop
      select
         when COUNT < SIZE =>
            accept WRITE(C : in ITEM) do
               POOL(IN_INDEX) := C;
            end;
            IN_INDEX  := (IN_INDEX mod SIZE) + 1;
            COUNT     := COUNT + 1;
      or
         when COUNT > 0 =>
            accept READ(C : out ITEM) do
               C := POOL(OUT_INDEX);
            end;
            OUT_INDEX  := (OUT_INDEX mod SIZE) + 1;
            COUNT      := COUNT - 1;
      end select;
   end loop;
end BUFFER;
```

A task equivalent to that given in the reference manual is obtained by the generic instantiation

```
task BUFFER is new BUFFERING(CHARACTER);
```

Use of the generic formulation permits the same strategy to be employed in a variety of different applications, for example:

```
task MESSAGE_BUFFER is new BUFFERING(ITEM is MESSAGE, SIZE := BACKLOG);
```

where MESSAGE is assumed to be a previously declared type and BACKLOG is some function which estimates a reasonable size for the buffer.

It is interesting to observe that the logic of the queuing strategy, shown by the example in the previous section, and that of the buffering strategy, presented above, are in many respects identical. The essential difference between the two approaches is that overflow and underflow are treated as exceptions in the former case, whereas in the latter case they merely result in some parallel task waiting until it can proceed.

13.3.4 A More Complicated Example

A final example, involving binary trees, is presented to illustrate the use of different kinds of generic program units in combination. A frequently encountered data type like binary trees is best encapsulated within a package, where the types of the leaves and nodes can be factored out as generic parameters. A straightforward definition of the (recursive) data structure in question might then be formulated as follows:

```
generic (type LEAF_TYPE;
         type NODE_TYPE)
package BINARY_TREES is

  type TREE is access
    record
      KIND : constant(LEAF, NODE);
      case KIND of
        when LEAF =>
          LVAL  : LEAF_TYPE;
        when NODE =>
          NVAL  : NODE_TYPE;
          LEFT  : TREE;
          RIGHT : TREE;
      end case;
    end record;

  -- specifications of standard operations on binary trees

end BINARY_TREES;
```

A number of standard operations associated with binary trees would normally be included within the generic definition module given above; for simplicity they will not be detailed here. Instead we will illustrate the typical ways in which binary trees are processed. These generally involve a recursive traversal (or *walk*) of the tree in one of a few characteristic orders (e.g., prefix order, infix order, or postfix order). These orders can be expressed as generic operations.

The most common of these orders is used in the example below. This is the *postfix walk*, where a certain operation is applied to each leaf, while another operation is applied to each node as well as to the results of previously processed left and right branches. The desired generic function might be defined (within the BINARY_TREES module) as

```
generic( type RESULT;
         function LEAF_OP (L  : LEAF_TYPE) return RESULT;
         function NODE_OP (N  : NODE_TYPE; L, R : RESULT) return RESULT)
function POST_WALK(T : TREE) return RESULT is
begin
  case T.KIND of
    when LEAF =>
      return LEAF_OP(T.LVAL);
    when NODE =>
      declare
        LB  : constant RESULT := POST_WALK(T.LEFT);
        RB  : constant RESULT := POST_WALK(T.RIGHT);
      begin
        return NODE_OP(T.NVAL, LB, RB);
      end;
  end case;
end POST_WALK;
```

Note that the recursive invocations of POST_WALK within this function cause no confusion (or infinite loop during instantiation) since the name of the generic function taken always refers to the same instantiation.

A number of useful utility functions on binary trees follow the pattern of a postfix walk. Some of these might well be included within the module BINARY_TREES itself. For example, COUNT, DEPTH and WIDTH (given appropriate definitions of the function ONE matching LEAF_OP, and of the functions SUM, SUM_PLUS_ONE and MAX, matching NODE_OP) are instantiated by the declarations

```
function COUNT is
  new POST_WALK(RESULT is INTEGER,  LEAF_OP is ONE,  NODE_OP is SUM_PLUS_ONE);

function DEPTH is
  new POST_WALK(RESULT is INTEGER,  LEAF_OP is ONE,  NODE_OP is MAX);

function WIDTH is
  new POST_WALK(RESULT is INTEGER,  LEAF_OP is ONE,  NODE_OP is SUM);
```

The advantages of using the generic facility in this fashion to formulate a basic pattern for several similar definitions are obvious. Another application of such definitions involves the use of binary trees to represent simple arithmetic expressions, where the leaves are integer values and the nodes correspond to the usual operators:

```
type OPERATOR is (ADD, SUB, MUL, DIV);
```

The appropriate definition can be obtained by instantiating the generic package

```
package EXPR_TREES is
  new BINARY_TREES(LEAF_TYPE is INTEGER,  NODE_TYPE is OPERATOR);
```

In an application, a use clause would be provided for this package and, for convenience, the tree type would be renamed by a subtype declaration:

```
use EXPR_TREES;
subtype EXPR is EXPR_TREES.TREE;
```

One may then introduce the specific operations associated with the type of tree in question. The most obvious is the evaluation function

```
function EVAL(E : EXPR) return INTEGER;
```

This, however, exactly follows the pattern of a postfix walk, and may therefore be directly obtained by instantiation

```
function EVAL is
  new POST_WALK(RESULT is INTEGER,  LEAF_OP is VALUE,  NODE_OP is INTERPRET);
```

where the requisite definitions of VALUE and INTERPRET are as follows:

```
function VALUE(I : INTEGER) return INTEGER is
  pragma INLINE;
begin
  return I;
end;

function INTERPRET(OP : OPERATOR; L, R : INTEGER) return INTEGER is
  pragma INLINE;
begin
  case OP of
    when ADD  => return L + R;
    when SUB  => return L - R;
    when MUL  => return L * R;
    when DIV  => return L / R;
  end case;
end;
```

Once again, the desired function is obtained by merely providing the appropriate operations for each leaf and node, while the details of the recursive treewalk are encapsulated within the generic function POST_WALK.

The binary tree example of this subsection presents a rather sophisticated structure, namely a generic recursive function (the function is recursive but there is of course no recursive instantiation), the declaration of which is itself nested within a generic package! While this example shows why such complicated formulations are occasionally desirable (see also [VH 75]), a word of warning is in order, particularly with regard to generic modules. Dependency between generic modules in the form of mutual instantiation is not allowed since such a structure would yield an infinite loop during instantiation:

```
generic(...)
package A is
   ...
end;

generic(...)
package B is
   ...
end;

package body A is
   ...
   package NEW_B is new B(...);     -- EITHER THIS IS ILLEGAL!!!
   ...
end A;

package body B is
   ...
   package NEW_A is new A(...);     -- OR THIS IS ILLEGAL!!!
   ...
end B;
```

13.4 Rationale for the Formulation of Generics

Several simplifications have been adopted in the design of a generic facility. Naturally when attention is focused on any given language feature, there is a tendency to believe that it is the most important feature. On the other hand, when trying to integrate such a feature into a language, one must take care not to make its impact on the language larger than its real importance to the user. This general argument applies particularly to generics. A generic facility is certainly useful, but its inclusion in the language should not be done at the expense of other important criteria, such as reliability, efficiency, and above all, simplicity of use.

The generic facility is expected to serve for the construction of general purpose parameterized packages. Whereas such packages are likely to be utilized by large classes of users, it should be realized that relatively few users will actually be involved in writing generic packages. Accordingly, we have tried to design a facility that can almost be ignored by the vast majority of users. They must indeed know how to instantiate a generic package and this is fairly easy. On the other hand, they need not be familiar with the rules and precautions necessary for writing generic program units.

A major simplification, in this respect, is achieved by adopting an approach based on a context dependent extension of the traditional techniques of macro-expansion, in preference to more complex facilities such as mechanisms for parameterization of data types. This solution has the advantage of introducing only minimal extensions to the language and its translators, and it is well implementable within the state of the art. It does not complicate the concept of type as perceived by the vast majority of users, and it nevertheless provides the flexibility required by the applications.

The goal of simplicity of use stated above has important consequences on the specification of formal generic parameters. The other major simplifying assumptions made in this language are the absence of any implicit derivation of attributes of type parameters (although default attributes can be specified), and similarly the absence of any implicit instantiation of generic program units. These issues will be discussed separately below.

13.4.1 Explicit Instantiation of Generic Program Units

An important simplification for the translation of program units obtained by generic instantiation is the requirement that such instantiation be explicit.

The approach taken here clearly distinguishes between the *instantiation* of a program unit derived from a generic program unit, and the *invocation* of a unit (calling a subprogram, using a module). Thereby it emphasizes the difference between translation time substitutions of generic parameters and execution time passage of actual parameters to subprograms. This provides a well-defined locus for the point of instantiation (and for reporting any errors arising from inconsistent substitution) while permitting the resultant program unit to be subsequently invoked as often as required, with the same degree of power and security as for any other non-generic program unit. This is a consequence of the fact that, once a generic declaration has been instantiated, the particular instance is indistinguishable from a similar program unit defined explicitly at the point of instantiation.

An alternate solution considered was implicit instantiation. For the purpose of the discussion of the complexity of implicit instantiation, consider the following generic function (which is actually just a different way of writing the power function in section 13.3.1):

```
generic( type T;
         function "*" (X, Y : T) return T);
function "**" (A: T; N : NATURAL) return T is
begin
   if N = 1 then
      return A;
   else
      return A * A**(N-1);
   end if;
end "**";
```

If implicit instantiation were provided then for

```
R   : RATIONAL;
I   : INTEGER;
```

exponentiation could be applied without prior explicit instantiation. Thus

```
R**5
I**5
```

would both be legal. The actual type used for T would have to be implicitly derived from the actual argument supplied for A (that is, RATIONAL for R, INTEGER for I).

It is quite clear that implicit instantiation would considerably complicate the algorithm used for identification of overloaded subprograms. For example, ** would be an overloading in the RATIONAL case whereas it would be a redefinition of predefined exponentiation in the INTEGER case. Moreover if the programmer had defined his own version of ** within the package RATIONAL_NUMBERS itself, for example:

```
function "**" (A : RATIONAL; N : INTEGER) return RATIONAL;
```

then this explicit definition would hide the generic definition in an application such as R**5. Thus the generic definition would be visible for some types and hidden for others.

Another problem arises for the correct identification of ** in the body of the generic unit itself: is it a recursive implicit instantiation or a recursive call of the same instance? In the simple example considered, it could easily be concluded that it is a recursive call. However, in general, it is not at all clear that the problem can always be resolved by a static analysis of the program (unless restrictions are adopted). A sufficient condition to guarantee that no generic operation will ever require an unbounded number of implicit generic instantiations during execution has been given in [BJ 78]. However such checks require a quite complex analysis of the program.

In conclusion, we consider implicit instantiation as a major research subject at this date (1979). The only solution within the current state of the art is explicit instantiation and this is hence the solution chosen for this language.

Explicit instantiation certainly requires more writing on the part of the user, but this has the effect of making him aware of what he is actually doing and thus contributes to reliability and readability. In addition, it offers distinct advantages in terms of efficiency, since the translator can easily identify the existing instantiations and, in some cases, perform optimizations such as the sharing of code for *closed* procedures.

13.4.2 Specification of Formal Generic Parameters

As stated earlier, it should be possible for a user instantiating a given generic package to completely ignore the internal details of this generic unit. In particular if any error is made in instantiating a generic unit, it should be reported to the user in terms of the generic instantiation itself and not in terms of the internals of the generic unit. This requirement has consequences on the form used for specifying formal generic parameters.

By analogy consider what is done for subprograms. For a normal (that is, non-generic) procedure, specification of parameters enables independent checks of the procedure body on the one hand and of the procedure calls on the other hand. Both must conform to the formal parameter specifications and these conformity checks can be done independently.

The specification of generic parameters provided by a generic clause must enable the same degree of independence.

(a) It should be possible to check that the text of the generic body is consistent with respect to the parameter specifications.

(b) For a given generic instantiation, it should be possible to check that the actual parameters conform to those specifications.

(c) The precision of the specifications should be sufficient to guarantee that if (a) is satisfied then any instantiation (b) will produce a semantically correct expanded program unit.

The solution adopted achieves these goals in a simple manner. For a type specified as restricted in a generic clause, no operation on the type is assumed available. For other types, only assignment and the predefined comparison for equality or inequality are available. Hence a formal parameter that is a type is considered as a private type within the generic unit. Any operation (apart from the above mentioned predefined operations) applied to objects of the type within the generic unit must be supplied as an additional generic parameter.

When checking the body of the generic unit, the generic clause thus provides the information required for the identification of all operators. When checking a given instantiation, it must match the generic clause and incorrect actual parameters can be reported. These two checks can be performed independently.

As an example consider the generic clause given for the function POWER:

```
generic ( type T;
          function "*" (X, Y : T) return T)
function POWER(A : T; N : NATURAL) return T is
begin
   if N = 1 then
      return A;
   else
      return A*POWER(A, N-1);
   end if;
end POWER;
```

The operation * is explicitly provided as a generic parameter as well as the type T itself. The parameter A and the result of the function POWER are both specified as being of this formal type. Thereafter the identification of the * appearing within the generic body in A*POWER(A, N-1) can be done as usual; it refers to the * declared in the generic clause. Similarly (implicit instantiation being impossible) the recursive call of POWER can be correctly identified. Hence the generic body can be completely checked.

Similarly a generic instantiation such as

```
function "**" is new POWER(RATIONAL, "*");
```

can be fully checked. It is correct if there exists an operation "*" on the type RATIONAL. The specification of this operation corresponds to

```
function "*" (X, Y : RATIONAL) return RATIONAL;
```

hence it matches the specification of the generic formal parameter. Conversely consider

```
function "**" is new POWER(RATIONAL, "not");  -- ILLEGAL!
```

This generic instantiation can be reported as incorrect since there is no operation **not** corresponding to the specification

```
function "not" (X, Y : RATIONAL) return RATIONAL;  -- ILLEGAL!
```

An alternative considered in this design was the implicit derivation of operations that are attributes of a generic type. The reasons for rejecting this alternative are similar to those leading to the rejection of implicit instantiation. If implicit derivation of attributes were allowed, the previous example could be rewritten with the generic clause

```
generic(type T)
```

and we would be left with the problem of identifying the * operation used in the body. For a given instantiation, say with the type RATIONAL, should the * operation be identified as a global operation in the context of the generic declaration or in the context of the generic instantiation? The two alternatives may lead to different results.

Note also that the identification may be ambiguous since both

```
function "*" (X, Y : RATIONAL) return RATIONAL;
function "*" (X : RATIONAL; Y : INTEGER) return RATIONAL;
```

would be acceptable in the absence of a specification for the formal operation *.

In general, the specifications of the identified operations could be quite different from instantiation to instantiation depending on the operations visible in the context of the instantiation. This could happen even if the type parameter is the same. As an example consider the expression A*(B/C) where A, B, and C are of type RATIONAL and where / is declared to deliver a result of type RATIONAL in one context and of type INTEGER in another context.

To summarize, implicit derivation of attributes would introduce an awkward context dependence and would require a complete checking of the generic body for each instantiation. This last consequence would be particularly unfortunate, since generic bodies could not be checked and proved correct independently of the context. It would defeat the goal stated initially, since some error messages would have to be stated in terms of what is done within the generic body.

For these reasons we have retained the solution of explicit specification of all operations that are used on objects of a formal type. This solution permits independent checking of generic units and of generic instantiations. Hence it fulfills our goal of permitting the user to ignore the internal details of the generic units instantiated in his programs.

13.4.3 Default Generic Parameters

As stated before, all operations applicable to a formal type must be specified explicitly in the generic clause. Nevertheless in order to keep generic instantiations as simple as possible, a facility for specifying default values for generic parameters is offered, as for normal subprograms.

In many cases, such default values will actually be expressed as attributes (predefined or not) of the formal type. For example, the generic clause of the function POWER can be rewritten as follows:

```
generic( type T;
         function "*" (X, Y : T) return T is T."*")
```

specifying T."*", the * attribute of the type T, as the default to be used in the absence of an explicit actual parameter. This parallels exactly the treatment of in parameters with default values for subprograms. The default parameter is optional and an instantiation such as

```
function "**" is new POWER(RATIONAL);
```

is taken as equivalent to the generic instantiation

```
function "**" is new POWER (RATIONAL, RATIONAL."*");
```

or to the generic instantiation

```
function "**" is new POWER(RATIONAL, "*");
```

This produces a valid declaration since * is defined for RATIONAL. For the same reason

```
function "**" is new POWER(BOOLEAN);
```

is an error since BOOLEAN."*" is meaningless (unless of course * were overloaded for BOOLEAN!).

Again, the generic body and the generic instantiations can be checked independently. Furthermore, the default can always be overriden by providing an explicit parameter as in

```
function "**" is new POWER(VECTOR, MULT);
```

Default attributes may conveniently be used to treat a type parameter as if it were a discrete type by providing the corresponding one to one mappings to the integers. Consider a unit that has a generic clause like

```
generic( type DISCRETE;
         FIRST  : DISCRETE := DISCRETE'FIRST;
         LAST   : DISCRETE := DISCRETE'LAST;
         function ORDINAL(X : DISCRETE) return INTEGER is DISCRETE'ORD;
         function VALUE(Y : INTEGER) return DISCRETE is DISCRETE'VAL)
package P is
   ...
end;
```

The integer equivalents of the values of type DISCRETE can always be used by applying ORDINAL and VALUE, for example, whenever a discrete range is required in loops, and so on. The package P can be instantiated for discrete types with a default derivation of attributes:

```
new P(CHARACTER)
```

Note finally that it may also be instantiated for a type T that is not discrete.

```
new P(T, FIRST_T, LAST_T, T_TO_INTEGER, INTEGER_TO_T)
```

with appropriately defined functions.

To summarize, the necessity to be able to check a generic body independently of its generic instantiations (an important user requirement) lead us to specify explicitly all operations applicable to a formal type in the generic clause. This has the effect of increasing the number of generic parameters that must be supplied and could hence lead to a heavy syntax of generic instantiations. However, one can specify default values for these operations, thus restoring the simplicity of generic instantiations.

In most applications, it should be possible to have only types as mandatory parameters and to provide default values for all operations. An example of such application is given by the generic input output package defined in chapter 14 of the Reference Manual. In the design of such a package it was vital to be able to simplify the instantiation (what the user needs to do) to the extreme. This has been achieved by providing default values for all generic parameters except, of course, for the type for which the package is instantiated. This is consistent with the goal stated in the introduction: writing a generic unit may well require some care; using it, on the other hand, should be extremely simple.

14. Representation Specifications and Machine Dependencies

There is an inherent dilemma in the design of a high order language with a system programming capability. On the one hand we are trying to achieve reliability by raising the level of the language. For example we provide data types and encourage the use of an abstract view of objects in which they are known only by the set of operations applicable to them. Controlling the applicable operations enables the detection of incorrect usage.

On the other hand, system applications require the ability to stay rather close to the machine, and not only for efficiency reasons. For example, defining a hardware description must be done in terms of the physical properties, the bit positions, etc. A mapping different from that prescribed by the hardware would be not only inefficient, but also incorrect and would not work at all. To produce a correct program in such cases we are forced to abandon the abstract view and to work in terms of the physical representation. This contradiction cannot be avoided; the language must permit dealing with objects at two different levels, the logical and the representational level.

Clearly, dealing with physical representations is inherently dangerous. However, some control can still be achieved if the language enforces a clear separation of the logical properties and their representation.

This separation principle is discussed below, along with the problem of changing representation and with the analysis of the issues raised by the different forms of representation specifications available in the language. This chapter also covers the means to specify the parameters of a configuration and their counterpart, environment enquiries. Finally, we present the means available for interfacing with other languages.

14.1 The Separation Principle

Several languages have already adopted the principle of separation of logical properties from representational properties. We may summarize this principle as follows:

Data type definitions are made in two steps:

(1) First, the logical properties of data are defined. They describe all the behavioral properties that programmers need to know. All algorithms are formulated in terms of these logical properties and are not based on knowledge of the representation.

(2) Second, the representation (implementation) properties of data may either be explicitly specified by the programmer or, in the usual case, chosen by default by the translator.

There are many advantages to such a separation. The most fundamental is the conceptual simplicity of formulating an algorithm in terms of abstract objects; the ability to abstract from a particular representation leads to clearer and better structured programs.

To certify correctness of a program working with data D and representation R, we have two disjoint proofs. First we show that the program is correct given the definition of D. Then we show that R is a correct implementation of D. Such a separation also ensures that users do not make hidden assumptions about the representational properties of data.

If at a later stage an alternate representation is used for the data, for instance for minimization of storage space, the formulation of the algorithms need not be modified. Conversely, when the algorithmic part of a program is modified, only the algorithm proof needs to be redone.

Another advantage is textual simplicity. In some cases, the representation of a data type may be dictated by external considerations such as the form of a hardware interface. The description of the representation may then have a complexity dictated by this external interface. However, by keeping the logical description textually distinct from that of the representation, we shall be able to retain the cleanliness of the logical description.

The above separation principle is reflected in the Green language by a clear textual separation between the declaration of the logical properties of data and the specification of their representational properties. The latter, called a representation specification, must appear after the declaration of the logical properties. Representation specifications are themselves grouped. Hence the parts of a program that might be hardware specific are easy to identify.

14.2 Types and Data Representation

In a language containing strong typing facilities, it is important to associate representation specifications with types rather than with individual objects. The basic reasons are simplicity and uniformity. To associate representation specifications with individual objects could mean that these specifications have to be duplicated in many separate declarations and it might therefore be difficult to maintain consistency, especially after repeated modification.

Alternately it could mean that representations are named and that each object declaration mentions both a type name and a representation name. However such a solution would result in less readable programs. As a consequence, a simpler solution has been used in the Green language and a representation is associated with a type.

Associating representation with type localizes this specification in one place. The specification is then implicitly associated with all objects of the given type (constants, variables, formal parameters, etc.).

A further advantage of this approach is that, while each type has some representation, the user generally does not need to be concerned with it. Accordingly, user defined representation specifications are optional, to be used only in cases where some external requirements must be satisfied; in the absence of such a specification, the representation is determined by the translator. Generally, the translator will choose an efficient representation, but no particular default is guaranteed. Thus, even a slight change to a declaration may result in a completely different representation.

Representation specifications may be more or less complete; in some cases they fully specify some mapping, while in other cases such as packing specifications they merely provide the translator with criteria for the choice of a representation.

14.3 Multiple Representations and Changes of Representation

When a program has to deal with objects existing on an external medium, one is faced with the problem of multiple representations. For example, records may be stored in a packed form on a file, but a program may require rapid access to the record components when the information is processed and hence may require an unpacked form. This is a classic situation where one wants two different representations for the same objects.

Although the details of the alternate representations are not part of the logical properties, we will show with the following example that the knowledge of the existence of alternate representations is, itself, a logical property.

14.3.1 A Canonical Example for Changes of Representation

Consider the problem of converting data from one external medium into a form ready to be output onto another external medium. Both data objects belong to the same enumeration type, but have different representations, each of which is fixed by the outside world. The following program fragment gives a hypothetical formulation (not following the syntax of the Green language) for the required procedure:

```
procedure CONVERT is

   -- declarations of the logical properties:

   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   X, Y : DAY;


   -- representation specifications (not in the Green syntax):

   representation FORM_A of DAY is
      (MON => 1, TUE => 2, WED => 3, THU => 4, FRI => 5, SAT => 6, SUN => 7);
   representation FORM_B of DAY is
      (MON => 0, TUE => 1, WED => 2, THU => 3, FRI => 4, SAT => 5, SUN => 6);
   for X use representation FORM_A;
   for Y use representation FORM_B;

   -- end of representation specifications (in hypothetical syntax)

begin
   ...
   Y := X;
   ...
end CONVERT;
```

In trying to establish the correctness of the above procedure, one finds that the information contained in the logical declarations of X and Y do not suffice. It can only be concluded that X and Y are of type DAY. To complete the correctness proof (that conversion is properly effected), one is forced to look into the representation specifications, and hence to violate the separation principle mentioned earlier. We are thus lead to the conclusion that any attempt to hide the existence of multiple representations at the logical level ultimately leads to a violation of the separation principle.

14.3.2 One Type, One Representation Principle

As argued earlier, since the concept of type is in the language, it is natural and desirable to use type as a carrier for representation. The view adopted in the Green language is thus that a unique representation corresponds to each type. This results in a significant simplification, since the user does not have to think in terms of multiple representations for a single type.

The solution to the problem of multiple representation is based on the declaration of types by means of derived type definitions. For example, a type B can be derived from a type A by declaring

```
type B is new A;
```

Since B derives its characteristics from A, both types have the same characteristics, for example the same components. However, they are distinct types and it is hence possible to specify different representations for A and for B. Change of representation can be achieved by explicit conversion between objects of type A and B since such conversions are defined for derived types. The derived type definition has the effect of creating a type with the same characteristics as another type, without rewriting its entire description (since that would define a distinct type for which no conversions are possible).

Note:

The one type, one representation principle must be understood in terms of the knowledge that the user has from the existence (as opposed to the details) of a representation. It means that if the user explicitly specifies the representation of a type, he may only specify one representation.

However, in cases where the representation is implicitly selected by the translator, it may use different internal representations in different contexts with full knowledge of the consequences of its choices.

14.3.3 Explicit Type Conversion and Change of Representation

The problem of change of representation is now straightforward; it can be expressed as an explicit type conversion between two logically equivalent types. A type conversion is specified by the use of a qualified expression, where the qualifier is the name of the type to which the expression is to be converted, for example

```
Y := EXTERNAL_DAY(X);
```

in the conversion problem presented earlier. This may be properly expressed in the Green language as follows:

```
procedure CONVERT is

   -- declaration of the logical properties:

   type DAY is (MON, TUE, WED, THU, FRI, SAT, SUN);
   type EXTERNAL_DAY is new DAY;        -- a derived type

   X  : DAY;
   Y  : EXTERNAL_DAY;

   -- representation specifications for the two types:

   for DAY use
     (MON => 1, TUE => 2, WED => 3, THU => 4, FRI => 5, SAT => 6, SUN => 7);
   for EXTERNAL_DAY use
     (MON => 0, TUE => 1, WED => 2, THU => 3, FRI => 4, SAT => 5, SUN => 6);

   -- end of representation specifications

begin
   ...
   Y := EXTERNAL_DAY(X);
   ...
end CONVERT;
```

The correctness of this procedure can now be established without violation of the separation principle. First we have to show that the program is correct given the definition of X and Y: Initially X contains a value of type DAY; the expression EXTERNAL_DAY(X) is legal since the type EXTERNAL_DAY is derived from the type DAY and it converts X into a value of type EXTERNAL_DAY that is assigned to Y. Second, it must be shown that the representation given for DAY and EXTERNAL_DAY are correct.

The same simple strategy would be used in the previously mentioned case of conversions of a record structure between a packed representation and an unpacked representation:

```
type OBJECT is
   record
      -- declaration of the components of objects
   end record;

type EXTERNAL_OBJECT is new OBJECT;  -- a distinct type derived from OBJECT

X  : OBJECT;
Y  : EXTERNAL_OBJECT;

for EXTERNAL_OBJECT use packing;
 ...

X  := OBJECT(Y);                     -- unpack
 ...
Y  := EXTERNAL_OBJECT(X);            -- pack
```

14.3.4 Implementation of Representation Changes

Although they are limited to types which are conformable, since they are declared as logically equivalent, type conversions may in some cases be very costly. As an example consider a record type with variant parts.

```
type V is
   record
      D : constant BOOLEAN;
      case D of
         when TRUE  => I  : INTEGER;
         when FALSE => R  : REAL;
      end case;
   end record;

type W is new V;

X  : V;
Y  : W;

for V use ...
for W use ...
 ...
X := V(Y);
```

The implementation of the assignment X := V(Y); cannot be achieved as simply as for a normal record assignment. Rather it must be done on a field by field basis equivalent to the following program (apart from the restriction on assignment to the discriminant):

```
X.D := Y.D;        -- would be illegal if written so
case X.D of
   when TRUE  => X.I  := Y.I;
   when FALSE => X.R := Y.R;
end case;
```

Although complex, it is within the state of the art to produce such code but it is nevertheless costly on some computers (note that there might be variants within variants). Expressing such changes of representations as explicit conversions warns the user about the potentially high cost of such operations.

14.4 Presentation of the Data Representation Facility

All representation specifications have the same general syntactic form in the Green language. The data type for which the representation is given is surrounded by the reserved words **for** and **use** as follows

```
for TYPE_NAME use ...
```

For example, to specify a packed representation for boolean matrices, one would write:

```
type BIT_MAP is array (1 .. 100, 1 .. 100) of BOOLEAN;

for BIT_MAP use packing;
```

In general, for array and record types where one would like to minimize storage, but for which the exact mapping is immaterial, it is possible to use this packing specification. Consider the case of an array of a given component type. For each component the translator must allocate a storage field with a certain number of bits. There may also be some gaps (i.e. unused bit fields) between two consecutive components. The effect of packing is to instruct the translator to minimize such gaps. On the other hand, if the component type is itself an array or record type it may also contain internal gaps. These are unaffected by the packing specification given for the array type. Minimization of such gaps could be achieved by a prior packing specification given for the component type itself.

It is also possible to use a length specification to specify the size that should be used for objects of a given type. This may be used for cases in which a user wants to optimize access time to frequently used record components, without having to specify the entire record layout.

For task names a length specification can be used to provide an upper bound for the storage needed by one task; for example, if the task contains recursive procedure calls, dynamic arrays, or local access types. Note that such a specification does not dictate the actual allocation strategy used for tasks; it could still be dynamic (at initiation time) or static (at the time of elaboration of the task declaration). It only supplies information to this allocation strategy. (Note that this is one case where the representation specification is applied to something other than a type name).

For access types, a length specification provides the size of the storage space to be reserved statically when elaborating the access type declaration. This space is used for the collection of dynamically created records associated with a given access type. The collection associated with an access type which has such a length specification is allocated statically at scope entry time exactly as for an array. Hence it permits the use of access types with their notational and efficiency advantages (component selection is cheaper than array indexing for arrays of records) without paying the potential costs of a more dynamic allocation strategy such as heap storage management.

To define sufficient storage space there is a need to know the storage size necessary for one element. For an access type T, the attribute T'SIZE can be used for this purpose. For example the size of a collection of dynamic records LIST_ELEM large enough to contain *approximately* 2000 records can be expressed as follows:

```
type LIST_ELEM is access
  record
    VALUE   : INTEGER;
    SUCC    : LIST_ELEM;
    PRED    : LIST_ELEM;
  end record;

for LIST_ELEM use 2000*LIST_ELEM'SIZE;
```

The number of dynamic records is only known as an approximation since the storage allocator may need some space and also because records with variant parts may have different lengths.

A length specification can also be used to achieve a biased representation for an integer type. For example, if we have a type ranging from 10_000 to 10_255, any value of this type can be represented in only 8 bits. Specifying a length of 8 bits for this type will result in the translator using a biased representation. For example:

```
type SKEWED is new INTEGER range 10_000 .. 10_255;
for SKEWED use 8;
```

14.4.1 Enumeration Type Representations

An enumeration type representation is used to specify the mapping from the elements of an enumeration type to the specific internal codes used to represent the elements.

The mapping is specified using an array aggregate in which the elements of the type are enumerated one by one. The type of such an aggregate is a one-dimensional array whose element type is integer and whose index range is the enumeration type itself. For example, consider the program that generates object code for a given machine and in which the operation codes for the machine are defined as an enumeration type. It is necessary to map the enumeration values into actual operation codes and this can be achieved as follows:

```
type MIX_CODE is (ADD, SUB, MUL, LDA, STA, STZ);

for MIX_CODE use
  (ADD => 1, SUB => 2, MUL => 3, LDA => 8, STA => 24, STZ => 33);
```

In this example the array aggregate is of type

```
array (MIX_CODE) of INTEGER
```

Obviously all enumeration values must be provided with distinct integer codes and their codes must be known at translation time. In addition, in order to get an efficient implementation for ordered enumeration types, the internal codes must follow the same ordering as the enumeration values. The order relations are then known through the internal codes, and there is no need for the translator to generate tables that contain the order relation.

As illustrated above, the specified internal codes do not need to be contiguous integers. We discuss the implications of this issue later in section 14.5.

14.4.2 Record Type Representations

Record type representations allow the specification of a storage layout for records. This representation is specified by giving the order of record components, their position, and their size in machine dependent terms. All the expressions defining such a specification must be static expressions; their values must be known at translation time. A global alignment clause can also be specified.

Storage Units:

The storage unit is a configuration dependent quantity representing the machine's quantum of storage. Its value can be accessed through the standard attribute SYSTEM'STORAGE_UNIT and is the unit of addressing implicitly used to denote the position of a component.

Bit Range:

A bit range is used to specify both the storage size in bits and the position of the first bit of a component inside a storage unit. The two expressions in the range represent the positions of the first and last bit respectively. This implies that the bit ordering inside a storage unit be known to the user. Such an ordering is configuration dependent and thus implementation defined. The logically first bit of a storage unit is always numbered 0. For example the specification

```
SYSTEM_MASK at 0 range 0 .. 7;
```

means that the component SYSTEM_MASK needs 8 bits of storage starting from the beginning of the storage unit. The storage size specified for a component must of course be large enough for the component. The translator shall check that it is compatible with the minimum needed for the representation of values of the component type.

Bit numbering extends through consecutive storage units; thus the specification

```
PROTECTION_KEY at 0 range 8 .. 11;
```

is legal even if the storage unit has eight bits on the machine considered.

At Clause:

The at clause specifies the position of a component by giving the position of the storage unit relative to which the bit range is counted. This position is itself relative to the first storage unit of the record, which is numbered 0. For example,

```
TRACK at 2 range 0 .. 15;
```

means that the component TRACK occupies 16 bits starting from bit 0 of the storage unit numbered 2. If the value of SYSTEM'STORAGE_UNIT were 8, the last bit of TRACK would actually be bit 7 in the adjacent storage unit numbered 3. Overlapping components are only allowed when they belong to distinct variants. Overlap of record components is not allowed within a variant, and the translator will check that this is not the case. For example the overlap of LINE_COUNT and CYLINDER in the following specification is legal since they belong to different variants:

```
type PERIPHERAL is
  record
    UNIT: constant (PRINTER, DISK, DRUM);
    case UNIT of
      when PRINTER =>
        LINE_COUNT : INTEGER range 1 .. 50;
      when others =>
        CYLINDER  : CYLINDER_INDEX;
        TRACK     : TRACK_NUMBER;
    end case;
  end record

-- assuming SYSTEM'STORAGE_UNIT = 8 bits

for PERIPHERAL use
  record at mod 4;
    UNIT        at 0 range 0 .. 7;
    LINE_COUNT  at 1 range 0 .. 7;
    CYLINDER    at 1 range 0 .. 7;
    TRACK       at 2 range 0 .. 15;
  end record;
```

When the record representation specification is incomplete, i.e. it does not specify the layout for all components, freedom is left to the translator to map the unspecified components in a way which is consistent with the logic of the record declaration. Translators shall produce object listings of record mappings upon request.

Alignment Clause:

When it is important that the object of a given record type be allocated on a given storage boundary, this can be specified by means of an alignment clause. The alignment is expressed as a number of storage units, and all addresses at which the objects are allocated must be exact multiples of the specified number of storage units (the address modulo the alignment expression must be zero).

14.4.3 Address Specifications

An address specification can be used to force the storage space of a given variable to be allocated at an address specified in storage units.

```
for PSD use at 16#40;
```

This form of specification can also be used for specifying the address of the code of a subprogram, or to link an interrupt with a given entry. The conventions used for mapping this integer on a hardware location are implementation dependent. As a consequence, other system dependent information can be derived from the value of this expression.

14.5 Enumeration Types with Non-Contiguous Representations

The specified internal codes of an enumeration type do not need to have contiguous values. This degree of generality is required, if character types are to be represented by enumeration types since many character sets have non-contiguous internal values.

We next discuss the implications of non-contiguous representations on assignment and comparison, indexing and case selection, and finally on iteration.

14.5.1 Assignment and Comparison with Non-Contiguous Enumeration Types

An assignment only results in moving a value from one location to another, and thus is not influenced by the non-contiguity of a representation. Similarly, non-contiguity has no impact on comparison.

14.5.2 Indexing and Case Statements with Non-Contiguous Enumeration Types

The simplest way to treat an array indexed by an enumeration type having a non-contiguous representation is to implement it like a normal array, and leave *holes* (i.e. unused positions) in the storage used for it. No conversion is then needed between the internal code and the real index to storage, since they have the same value. In a similar way, the internal jump table used for a case statement may have holes.

Note that no problem arises when such arrays are passed as parameters to subprograms since the index type is part of the array type and the same mapping can be used inside and outside the subprogram.

The user should be aware of the hidden storage costs involved. This is certainly preferable to prohibiting the use of types with non-contiguous representations for indexing and in case statements. If we consider character sets, for instance, the proportion of holes remains at an acceptable level.

14.5.3 Iteration Over Non-Contiguous Enumeration Types

We are faced with a more severe problem when a loop parameter ranges over the values of a non-contiguous enumeration type; simply incrementing the value of the loop parameter by a constant at each iteration will not work correctly! To keep the same underlying mechanism, we need the notion of a characteristic vector, which carries the information on the internal codes. For every such loop, the translator will include code to interrogate the characteristic vector.

This mechanism is illustrated by the following example. Consider the type MIXED, for which a non-contiguous representation has been specified.

```
type MIXED is (A, B, C, D);
for MIXED use (A => 30, B => 32, C => 33, D => 37);
```

A loop statement iterating over the values of the MIXED type can be written as follows:

```
for I in MIXED'FIRST .. MIXED'LAST loop
   P(I);
end loop;
```

The translator will produce an object code which is equivalent to the following text (apart from typing rules):

```
PRESENT : constant array (30 .. 37) of BOOLEAN
                := (30 | 32 | 33 | 37 => TRUE, others => FALSE);

for I in 30 .. 37 loop
   if PRESENT(I) then
      P(I);
   end if;
end loop;
```

As illustrated above, the translation involves a characteristic vector (PRESENT) which is used to generate the integer values corresponding to the enumeration type MIXED. Thus we see that iterating over such types is possible, but involves a small extra cost.

14.5.4 Character Types

Character types are a typical example of enumeration types with not necessarily contiguous representations. The predefined character type CHARACTER denoting the full ASCII character set of 128 characters is contiguous but the same is not true for other widely used character sets such as EBCDIC. Such character sets will generally be defined in library modules, including both the character type declaration and the associated representation specification. It may be convenient to provide such a definition in two steps. For example:

```
type CHAR is (enumeration_of_all_EBCDIC_characters);

type EBCDIC is new CHAR;  -- same characters as CHAR

for EBCDIC use (codes_corresponding_to_EBCDIC_characters);
```

A user to whom the internal code is relevant (e.g. because he is performing input-output) will declare objects of type EBCDIC. For other usages, especially if such characters are to be used as indices, in case statements and in iterations, the user might prefer to use the type CHAR. Since no representation specification is given for this type, the translator will adopt a default representation that is convenient for indexing and iteration. Explicit conversions between the two types can be performed for input-output.

14.6 Configuration Specification and Environment Enquiries

To generate object code, some machine and configuration dependent properties such as the machine model, memory size, and special hardware options must be available to the translator. Hence specification of configuration dependent features must be possible. Typical uses of such information are for the detection of resource usage overflow and the generation of special purpose instructions for the target machine.

Conversely, programs may need to access information that is known to the translator. There are numerous uses for such information. A user level input-output routine may need to invoke alternative algorithms depending upon the object machine configuration (with the discrimination being made at translation); similarly, it may need to know the size of the storage unit for the object machine, and the size of the objects transferred.

The approach used in the Green language is to have pragmas for the specification of information to the translator, and to use predefined attributes as general environment enquiries in order to refer to information known by the translator.

14.6.1 Pragmas

Pragmas serve to provide the translator with information that is relevant for the translation process. This information does not affect the semantics of programs. For example we may have

```
pragma SYSTEM(MULTICS);
pragma STORAGE_UNIT(36);
pragma INCLUDE(COMMON_TEXT);
pragma OPTIMIZE(SPACE);
pragma LIST(ON);
pragma PAGE;
pragma SUPPRESS(OVERFLOW);
```

We can distinguish two main categories of pragmas: configuration specifications and translator options. The configuration of a given object machine can be specified for a translation unit by a list of pragmas. Among them there will generally be one for the machine architecture model SYSTEM, one for the memory size MEMORY_SIZE, and one for the size of storage unit STORAGE_UNIT. This category can also include pragmas defining special hardware options, device configurations, and any special characteristics of an operating system.

Translator options will generally include time or space optimization, a control for the listing of source and object programs, and so forth.

14.6.2 Predefined Attributes

Predefined attributes may be viewed as the interrogative counterpart of pragmas. They provide an environment enquiry mechanism, which can be used to obtain configuration dependent information and more generally other information known by the translator. We have two major categories of environment enquiries, corresponding to the two main categories of pragmas.

Configuration characteristics can be accessed via configuration dependent constants expressed as attributes of the predefined name SYSTEM. For example we have SYSTEM'MEMORY_SIZE for the memory size and SYSTEM'NAME for the machine architecture model.

The translator options are accessed via predefined boolean attributes of the predefined name OPTION, for example:

```
OPTION'SPACE      -- TRUE if space is the current optimization criterion
```

The mechanism is also used as a general environment enquiry mechanism: for example, to access properties of program components, the bounds of an index of an array, the address of an object, or information about a record representation; to obtain the implemented range of a program component; to read the system's real-time clock; to obtain the length of the queue associated with an entry, etc.

As mentioned in the section on lexical issues, the names of predefined attributes are always preceded by an apostrophe. As a consequence the corresponding identifiers are not reserved. Some typical examples are given below.

```
TABLE'FIRST              -- the first index of TABLE
MATRIX'FIRST(2)          -- the first index of the second dimension of MATRIX
OLD_PSW'ADDRESS          -- the address in storage units of OLD_PSW
X.MASK'POSITION          -- the first position of the component MASK in X(at clause)
X.MASK'FIRST_BIT         -- the position of the first bit of MASK
X.MASK'LAST_BIT          -- the position of the last bit of MASK
INTEGER'SIZE             -- the implemented size of INTEGER in bits
REAL'RADIX               -- the implemented radix of REAL
REAL'EXPONENT_FIRST      -- the implemented lower bound and
REAL'EXPONENT_LAST       -- upper bound of the exponent for REAL
SYSTEM'CLOCK             -- the real-time clock
SYSTEM'STORAGE_UNIT      -- the number of bits in a storage unit
```

14.6.3 Configuration Specification and Conditional Compilation

Sometimes it is desirable to write a program in which portions vary according to the object machine configuration. Such *conditional* translation can be achieved by conditional statements selecting among alternative program fragments. For example, a program providing different algorithms for different systems may appear as follows:

```
pragma SYSTEM(MULTICS);
 ...
case SYSTEM'NAME of
  when TENEX =>
    -- part specific to TENEX
  when MULTICS =>
    -- part specific to MULTICS
  when UYK20 | AYK14 =>
    -- part specific to these systems
  when VS_370 =>
    -- part specific to VS_370
end case;
```

The system name established by the pragma is considered as an enumeration value that can be tested later. Since the system name is known at translation time, a translator may decide to optimize the case statement and generate only the code corresponding to the current system name. Thus the program can be tailored to a given machine.

This facility is deliberately primitive. Conditional translation of declarations can be achieved in a limited way by variant record types. On the other hand no general mechanism for conditional translation of program units is provided. We consider such mechanisms for text selection to belong to the domain of the facilities provided by the support tools for the language.

14.7 Interface with Other Languages

A limited facility for machine code insertions has been included in the Green language. This facility has the advantage of clearly isolating the use of machine language. However, its general use is heavier than direct use of an assembler.

Each machine instruction appears as a code statement, which is a record aggregate of a record type defining the corresponding instruction. Such record definitions will generally be available in a library package for each machine. The library package must also contain the representation of the record describing the machine instruction format. These code statements are used in procedures which must contain only code statements and are included inline.

The following example illustrates the use of a set system mask instruction on 370 like machines. This example shows that it may be necessary to use implementation specific predefined attributes in code statements, such as M'BASE (the base register used for M) and M'DISP (the displacement of M).

```
M : MASK

procedure SET_MASK is
  pragma INLINE;
  use INSTRUCTIONS_370;    -- makes visible the identifiers SI_FORMAT, SSM, etc.
begin
  SI_FORMAT(CODE => SSM, B => M'BASE, D => M'DISP);
end;
```

Additional implementation specific pragmas may be needed to specify the register and linkage conventions. Obviously such pragmas cannot be machine independent; the only order that may be brought by a high level language in such a matter is to standardize the mechanism by which such specifications are given. In the Green language this mechanism is provided by pragmas. As a final example, pragmas can also be used to specify that a subprogram is written in another language:

```
procedure GAUSS(A : in out MATRIX; X : in out VECTOR; N : INTEGER) is
begin
   pragma INTERFACE(FORTRAN);
end;
```

In this example a procedure skeleton is written in Green. For other subprograms it has the effect of specifying the calling convention in Green terms. The sequence of statements is limited to the INTERFACE pragma, informing the translator about the corresponding linkage conventions and also to expect the object code to be provided later (at linkage edition time). Of course, translators may impose restrictions on the form of parameters allowed. Not all translators need provide such a capability.

14.8 Unsafe Programming

The conversions allowed among numeric types and among types that are derived from each other are safe conversions that do not violate the rules of type checking.

Unsafe type conversions can be achieved in any language that permits code insertions or address specifications. Such conversions may, for example, be needed if a user wants to define his own allocation strategy for access types. In this case, conversions from integer to access values are necessary to define an ALLOCATE procedure, and conversely a FREE procedure.

From a programming management and also from a maintainability point of view, it is desirable to provide a standard way to achieve such unsafe conversions. In this way, parts of a program using such dangerous features are made easier to identify. The following library module is predefined to that effect.

```
package UNSAFE_PROGRAMMING is

   generic(type S; type T)
   function UNSAFE_CONVERSION(X : S) return T;

end UNSAFE_PROGRAMMING;
```

A program unit using unsafe type conversions must include this package in its visibility list. Hence its visibility restriction will appear as

```
restricted(..., UNSAFE_PROGRAMMING, ...)
```

In addition, it must instantiate the function UNSAFE_CONVERSION for the types for which the convention is desired. For example

```
function UNSAFE_INT_TO_LIST is new UNSAFE_CONVERSION(INTEGER, LIST_ITEM);
```

The programming environment may be able to control and restrict the programs that are allowed to get access to the package UNSAFE_PROGRAMMING.

15. Input-Output

15.1 Introduction

Input-output is recognized as a source of difficulty for programmers. It often accounts for a large part of a programming system and generally is either neglected in a language formulation (e.g. Algol 60, Coral 66) or causes many special and often confusing features to be built into a language (e.g. Input-Output lists and formats in Fortran, format variables in Algol 68). Even Pascal uses special procedure structures (with optional file parameters and a special field width separator) available only in the predefined input-output procedures.

Even more importantly, the needs for application level input-output may vary greatly between classes of applications. For example, file manipulation, batch processing, line and page layout, interactive input, and non-character processing pose significantly different problems. An attempt to build in special features to cover the range of input-output applications would mean that every user and every translator would be forced to take account of this additional complexity.

A major design goal in the Green language was therefore to provide the ability to develop a rich set of input-output facilities without additional language constructs. In this chapter we demonstrate that this ability has been achieved. Input-output packages can be written without resort to special features. Given this ability, it will be possible for user groups to develop and standardize specialized input-output packages corresponding to major application classes.

The language nevertheless provides a recommended set of input-output operations in compliance with the Steelman requirements and in recognition of the fact that the existence of such a set is essential for portability.

Three standard input-output packages are given in the language definition. The generic package INPUT_OUTPUT defines a general set of user level input-output operations. Additional operations for text input-output are defined in the standard package TEXT_IO. Finally the package LOW_LEVEL_IO defines the form of the operations used for dealing with low level input-output. The design of these packages is discussed in this chapter. We conclude by a discussion of the problems involved in writing an input-output package within the language itself.

15.2 General User Level Input-Output

The user level input-output operations are defined by the package INPUT_OUTPUT. These operations are applicable to files containing elements of a single type. After discussing the conventions used to designate external files, devices, and internal files we discuss the facilities offered by the package.

15.2.1 Designation of External files

Input-output operations effect data exchanges between a processor running on behalf of the user program and some peripheral device. Traditionally the notion of file, seen as a repository of information, is often distinguished from that of a device. However the logical behavior of a program does not depend on the source or destination of data, as long as the data represents the desired information. For example it is perfectly reasonable to interface the same program with a disk file at one time, and with a terminal at another time.

In consequence, the conventions used to designate the target of an input-output operation should not necessarily distinguish between file names, device names, volume names, etc. In addition these conventions should not conflict with those of any system on which the language is implemented. This dictates a flexible approach.

We define an *external file* as any component of a computer system that can be designated as the target of input-output operations. The name of an external file is written as a string and its interpretation is system dependent. In some implementations the name may contain additional control information.

15.2.2 Designation of Internal Files

An external file, as defined above, is an entity which can produce or absorb data. The lifetime of an external file is not tied to a particular program. For example an external file can be created by one program and be later deleted by another program.

In contrast, within a program performing an input-output operation, an object called an internal file (or just a file) is used to refer to the external file. Such files are variables of the encapsulated types IN_FILE, OUT_FILE, and INOUT_FILE. These types and the set of operations applicable to them are defined in the package INPUT_OUTPUT.

The most salient feature of a file is that it is associated with an element type: on any particular file, input and output operations are done in terms of values of that type. A file is also permanently associated with a *mode*: there are files on which only input is possible (IN_FILE), files on which only output is possible (OUT_FILE), and files on which both input and output are possible (INOUT_FILE).

Note that the mode is a property of a file as an object manipulated by the program, and not a property of a particular external file. Such restrictions may be placed on given devices or data sets by a given system, but defining them is not within the realm of the language, nor should it be. Indeed, different systems may have defined arbitrarily sophisticated protection schemes to control access to resources. The state of the art in the design of such schemes is constantly evolving, and standard primitives of a language should not interfere with this evolution.

Access to certain external files may nevertheless be limited because of such protection schemes, or because of other physical limitations. Two exception conditions correspond to these limitations: FILE_NAME_ERROR is raised when a CREATE or OPEN cannot be performed, and FILE_USE_ERROR is raised if an operation cannot be performed on an open file because of physical or logical limitations.

The mode of a program file is given by the declaration of a file variable, since it is implicitly associated with the possible types (IN_FILE, OUT_FILE, or INOUT_FILE).

15.2.3 Overview of File Operations

A file can be dynamically associated with a specified external file. The association is established by a CREATE or OPEN operation, and severed by a CLOSE or DELETE operation. A file that is associated with an external file is said to be open. The status of a file can be interrogated by the function IS_OPEN. The operations CREATE and OPEN can only be performed on files that are not open. All other operations must be performed on open files.

The name of the external file associated with an open file is given by the function NAME. The total number of elements of the file is given by SIZE. SIZE is applicable to external files of any kind: for a data set this can be computed from the storage size of the data set. For devices that perform stream input or output, SIZE corresponds to the number of elements read or written on that file by the program.

Once a file is open, a particular element is recognized as the target of the next input or output operation. The position of this component in the file is given by the function NEXT. The first component of a file corresponds to the position 1. The value returned by NEXT is automatically incremented by a successful input or output operation. It can also be altered explicitly by the procedure SET_NEXT. Certain restrictions on an external file may place limitations on the use of SET_NEXT, or prohibit it entirely. For example, it may be impossible to back up input from a terminal, or to set it outside a certain range on a card-reader. Note however that in the absence of other limitations, it is always possible to set the next position to a value larger than the file size, or to a non-positive value. An exception would be raised only upon a subsequent input or output operation. Unless forbidden by the nature of the external file, writing an element at a position beyond the current file size has the effect of extending the size.

The next position is always incremented after a successful read or write, independently of the mode of a file; thus all files can be accessed either sequentially or (if SET_NEXT is allowed) randomly. In particular, the user can conveniently perform a sequential scan of part of a file, then move to another part and again scan the file sequentially.

Actual input and output operations are performed by the READ and WRITE primitives. READ is not defined for OUT_FILE, and WRITE is not defined for IN_FILE. If further restrictions prohibit the operation, a FILE_USE_ERROR exception is raised. If an attempt is made to READ past the end of a file, an END_OF_FILE exception is raised. Explicit testing of an end of file condition can be easily done by

```
NEXT(F) > SIZE(F)
```

Note that the user level input-output facilities are provided without any reference to a particular buffering scheme. On a simple system a READ or WRITE operation could correspond to the appropriate physical input or output operation without any buffering. When needed, a buffering strategy can be defined by manipulating files whose components are arrays. For example, we may have

```
type BUFFER is array (1 .. 512) of INTEGER;
package BUFFERED_IO is new INPUT_OUTPUT(BUFFER);
```

15.3 Text Input-Output

Several input and output operations involve character strings. These operations are particularly common in programs that receive input from, or produce output for, a human operator. Such use is sufficiently widespread to justify provision of a more sophisticated set of primitives than those available for general files of characters.

Assuming a predefined character set, it must at least be possible for the user to request input or output of values of any primitive type represented as strings of characters. Such an effect can of course be achieved in two steps: in the case of input, first read a string of characters, and then convert it to an internal value; in the case of output, first convert a value to a string, and then write out each character of the string. Conversion procedures are actually provided for each scalar type of the language by the predefined attributes REP and VAL. It would however be unacceptable to force the user who wants to output, say, an integer X to write:

```
declare
   S : STRING := INTEGER'REP(X);
begin
   for I in 1 .. S'LENGTH loop
      WRITE (S(I));
   end loop;
end;
```

Therefore it is necessary to predefine a standard function (say PUT) which does at least that. The need for procedures that combine input or output with string conversion is even more acute when reading values, since strings of a particular form must be recognized before being converted.

The operations defined in the package TEXT_IO consider a text as a stream of characters. Lines are particular logical units of arbitrary length. This view corresponds to interactive systems practice. In addition, a simple convention allows the user to fix the line length, and to use the same primitives for fixed format input or output.

Consistent with the design of all user level input-output, the text input-output facilities are defined entirely within the language. No special language construct (e.g. formats, pictures) are introduced. In particular, no special syntax rule is added. Input and output facilities are defined for all scalar types, including user defined enumeration and fixed point types.

In order to minimize the number of names that must be remembered by a user, the basic operations for text input-output appear as overloaded subprograms called GET and PUT.

In order to simplify the writing of input-output operations, default parameters have been used for specifying format information. In addition the package defines a standard input file and a standard output file, both opened during the elaboration of the package body. Each PUT or GET operation is defined in two overloaded versions: one where an explicit file must be specified, and one where it is not specified. The latter versions operate on the default input file (for GET) and the default output file (for PUT). The initial default files can be accessed explicitly by the functions STANDARD_INPUT and STANDARD_OUTPUT. The user may change the default input and output files with the procedures SET_INPUT and SET_OUTPUT. Since STANDARD_INPUT and STANDARD_OUTPUT are functions, the user cannot close them (since CLOSE and DELETE expect an out parameter).

Note that the default parameter mechanism has not been used for the definition of the alternative versions of GET and PUT (for an explicit or for a default file) since such a formulation would not permit changing the default file.

15.3.1 Characters, Lines, and Columns

The package TEXT_IO is defined for streams of characters of the ASCII character set. This character set includes a set of control characters, and is especially adapted to free format text. A consequence is that the notion of line need not be defined in terms of a fixed length. Indeed, a line can be viewed as any sequence of characters enclosed between two line marks. A line mark can be the beginning or the end of the file, or an implementation defined sequence of characters (for example the sequence carriage return, line feed). Line marks can be denoted by the predefined string NEWLINE.

The notion of current line can be associated with the number of line marks that have been input or output. The effect of SET_LINE on input will thus be to read forward or backward over the appropriate number of line marks (equal to the difference between the target line and the current line if the target line number is larger, or otherwise to the difference between the current line and the target line, plus one). This latter case may be prohibited. If not, the backward read is followed by a forward read over the last line mark, to get to the beginning of the line. On output, the effect of SET_LINE is device dependent: it may either have the same effect as on input (e.g. with disk files), or output the appropriate number of line marks (e.g. on a terminal).

Note that SET_LINE always has the implicit effect of setting the current column to the beginning of the line. This decision (rather than to set the current line independently of the current column) corresponds to a more common usage. In addition setting the current line alone would not be clearly defined with variable length lines, since the current column may end up being off the end of the current line.

The notion of column is associated with the position of a character from the beginning of a line. The effect of special control characters is defined so as to preserve as much as possible the identity between the column number and the position of a character on a printed line. The primitive SET_COL can be used to skip to a certain position on a line. On some output devices, it may have the effect of spaces, or backspaces.

Although these primitives are defined for variable length lines, it is fairly easy to apply them to fixed length lines. The user can first set the line length on a particular file to a value of his choice. When this is done, line marks are implicitly inserted after the printable character in column L, where L is the line length. With these layout primitives, the user can define procedures such as:

```
procedure EJECT( FILE : OUT_FILE; N : INTEGER := 1) is
begin
   SET_LINE(FILE, LINE(FILE) + N);
end;
procedure EJECT(N : INTEGER := 1) is
begin
   SET_LINE(STANDARD_OUTPUT, LINE(STANDARD_OUTPUT) + N);
end;
procedure SPACES(FILE : OUT_FILE; N : INTEGER := 1) is
begin
   SET_COL(FILE, COL(FILE) + N);
end;
procedure SKIP( FILE : IN_FILE; N : INTEGER := 1) is
begin
   SET_COL(FILE, COL(FILE) + N);
end;
```

15.3.2 Text Processing and Formatting

The string representation of an internal value is defined in accordance with the lexical representation of literals in the Green language. Note that it may be necessary, upon input, to look two characters ahead to determine the end of a lexical unit. For example, if the input text contains the string

12.83ERGS

then, when a GET is issued for a floating point value, it is necessary to see the R of ERGS to determine the result of GET, as the E could be the beginning of an exponent.

On the other hand, with the convention to output a space in front of any numeric or enumeration value, it is possible to have a one-to-one mapping between the operations needed to write a text, and those needed to read it.

The GET and PUT operations can be used for both free and fixed format texts. With free format the user is primarily interested in outputting a value without extraneous characters, whereas in fixed format, the goal is to place the value at a specified position on a line. The default parameter mechanism of the Green language can be used to achieve both effects in a single definition, by providing a field width argument whose default value (0) will cause the minimum number of characters to be used. Thus, omission of the width parameter provides free format output, whereas fixed format can be obtained when it is given. This can be contrasted with the Pascal convention which provides an implementation defined default field width, thus making it awkward to output small numbers in free format.

In the case of fixed and floating point types, the number of digits appearing after the decimal point (*the length of the mantissa*) is by default the minimum number of digits consistent with the accuracy specified for the type.

The PUT procedure can be defined for all scalar types in terms of the predefined attribute REP, the procedure PUT for strings, and two functions, LEFT_PAD and RIGHT_PAD, defined by:

```
function LEFT_PAD(S : STRING ; N : INTEGER) return STRING is
   N_BLANKS : constant INTEGER := N - S'LENGTH;
   T : STRING(1 .. N_BLANKS) := (others => ' ');
begin
   if N_BLANKS <= 0 then
      return S;
   else
      return T & S;
   end if;
end LEFT_PAD;

function RIGHT_PAD(S : STRING ; N : INTEGER) return STRING is
   N_BLANKS : constant INTEGER := N - S'LENGTH;
   T : STRING(1 .. N_BLANKS) := (others => ' ');
begin
   if N_BLANKS <= 0 then
      return S;
   else
      return S & T;
   end if;
end RIGHT_PAD;
```

For example, the integer form of PUT can be defined as

```
procedure PUT(FILE : OUT_FILE; ITEM : INTEGER; WIDTH : INTEGER := 0) is
   S : STRING := INTEGER'REP(ITEM);
begin
   if COL(FILE) + MAX(S'LENGTH, WIDTH) >= LINE_LENGTH(FILE) then
      SET_LINE(FILE, LINE(FILE) + 1);
   elsif COL(FILE) /= 1 then
         PUT(" ");
   end if;
   PUT(FILE, LEFT_PAD(S, WIDTH));
end;
```

In the case of fixed point types, the internal representation is dependent on the declared range and delta. Therefore, in contrast to floating point types, no predefined input-output function can be given without knowledge of the type. A similar case exists for enumeration types since the corresponding enumeration literals are not known in the TEXT_IO package. The solution taken provides PUT and GET procedures in generic packages that can be instantiated by the user. Default generic parameters reduce the complexity of the instantiation by automatically providing the formatting functions REP and VAL.

Consider the following example

```
type FRAC is delta 0.005 range -1000.0 .. 1000.0;
type COLOR is (RED, YELLOW, GREEN, BLUE);

package FRAC_IO is new FIXED_IO(FRAC);
package COLOR_IO is new ENUM_IO(COLOR);

declare
   use FRAC_IO, COLOR_IO;
   F   : FRAC := 1.990;
   C   : COLOR := RED;
begin
   PUT(C); PUT(F);
   PUT(NEWLINE);
   PUT(C, 6); PUT(F, 12, 5);
end;
```

In the output below, note that an enumeration value is left justified, whereas a number is right justified:

```
RED 1.990
RED          1.99000
```

The default mantissa for FRAC is determined upon instantiation, by the expression

```
DELTA_REP'LENGTH - 2
```

where

```
DELTA_REP = FRAC'REP(FRAC'DELTA - INTEGER(FRAC'DELTA))
```

Since FRAC'DELTA is 0.005, DELTA_REP will be "0.005" and the expression thus evaluates to the desired 3 digits. The complexity of the expression is needed to cope with deltas greater than 1, in which case the integer part must be wholly represented.

Although the primitives defined by INPUT_OUTPUT are not given explicitly in the TEXT_IO package specification, those taking IN_FILE and OUT_FILE parameters are inherited by the derived types declared in TEXT_IO.

15.3.3 An Example

Consider the problem of writing a printout on a line-printer, from information found in the files ARTICLES (text) and INVENTORY (integer). Any error must be reported on the user terminal.

```
procedure LISTING is
   use TEXT_IO;
   package INT_IO is new INPUT_OUTPUT(INTEGER);
   PRINTOUT    : OUT_FILE;      -- i.e. TEXT file
   ARTICLES    : IN_FILE;       -- also TEXT
   INVENTORY   : INT_IO.IN_FILE;
   CH          : CHARACTER;
   VALUE       : INTEGER;
   POS         : INTEGER;
begin
   OPEN(PRINTOUT, "prtr");    -- recognized as line printer
   OPEN(ARTICLES, "articles");
   INT_IO.OPEN(INVENTORY, "inventory");
   SET_LINE_LENGTH(PRINTOUT, 30);
   SET_OUTPUT(PRINTOUT);    -- printout is now default output file
   PUT("article");
   SET_COL(PRINTOUT, 20);
   PUT("inventory" & NEWLINE & NEWLINE);
   loop
      begin
         INT_IO.READ(INVENTORY, VALUE);
         POS := 1;
         loop
            GET(ARTICLES, CH);
            exit when CH = " ";
            if POS < 20 then
               PUT(CH);
               POS := POS + 1;
            end if;
         end loop;
         SET_COL(PRINTOUT, 20);
         PUT(VALUE, 9);    -- fills line entirely (new line will be automatic)

      exception
         when END_OF_FILE =>
            CLOSE(PRINTOUT);
            CLOSE(ARTICLES);
            INT_IO.CLOSE(INVENTORY);
            SET_OUTPUT(STANDARD_OUTPUT);
            exit;
         when others =>
            PUT(STANDARD_OUTPUT, "printout error at line");
            PUT(STANDARD_OUTPUT, LINE(PRINTOUT));
            PUT(STANDARD_OUTPUT, NEWLINE);
            PUT(NEWLINE);   -- on printout
      end;
   end loop;
end LISTING;
```

15.4 Low Level Input-Output

Low level input-output facilities are especially needed in embedded computer systems, since signal processing and interaction with non-standard peripheral devices are common. Clearly, major system dependencies cannot be avoided in this area. At best the language can provide a set of standard calling conventions for dealing directly with peripherals. The specific device and data description cannot however be given.

Interaction with peripheral devices involves three forms of actions: starting an operation on a device, interrogating the status of a device, and waiting for completion of an operation. Facilities to deal with this latter case are provided by the entry mechanism and interrupt specification. The first two cases, however, constitute requests from the program. For these, two procedure names are introduced: SEND_CONTROL to start an operation, and RECEIVE_CONTROL to interrogate the status. Both take two arguments: DEVICE identifies a particular peripheral device, and DATA corresponds to the information that should be exchanged with the device (hence DATA is an **in out** parameter).

For the definition of such procedures, we are faced with two problems: efficiency and generality. Efficiency dictates that an operation which normally requires a small number of machine instructions should not be surrounded by lengthy checks. This could be achieved by making the low level primitives built-in to a given compiler. However generality requires the ability to write the appropriate SEND_CONTROL and RECEIVE_CONTROL operations whenever a new device is added to the system, without forcing a recompilation of the translator. Hence these operations cannot be built-in.

In order to satisfy these apparently conflicting goals, subprogram overloading and code statements can be used. As many *device types* should be introduced as required by the interfacing conventions of the system. Similarly, for each device type, appropriate *data types* should be introduced. For each meaningful combination of device type and data type, overloaded definitions of SEND_CONTROL and RECEIVE_CONTROL can be given, and the corresponding subprogram bodies may use appropriate code statements. The general form of the package LOW_LEVEL_IO is as follows

```
package LOW_LEVEL_IO is
   -- declarations of different device-types
   -- declarations of different data-types
   -- declarations of overloaded procedures for these types:
   procedure SEND_CONTROL    (DEVICE : device_type; DATA : in out data_type);
   procedure RECEIVE_CONTROL (DEVICE : device_type; DATA : in out data_type);
end LOW_LEVEL_IO;
```

Thus if a user must introduce a new device, then the appropriate types and procedures can be defined independently of existing ones; this only requires a recompilation of the package LOW_LEVEL_IO.

15.5 Writing an Input-Output Package in the Language

The motivations for defining user level input-output within the language itself were given in the introduction. Defining input-output operations is not inherently difficult. On the other hand defining file types and their operations by the normal mechanisms of type definitions and without sacrificing reliability and efficiency is a major challenge.

Most languages so far have avoided the difficulty by having built-in file types. We believe this approach to be wrong, since the problem raised by the definition of specialized application data types will be of the same order of difficulty as those raised by files. Thus the inability to define files within the language would be a symptom that should not be neglected by the language designer.

The solution taken defines files as restricted private types, one type for each file mode. This enables us to enforce control over file manipulation in the language itself. Since the type is a static property, it is possible to check at compilation time that a READ operation is not performed on an OUT_FILE; similarly that a WRITE operation is not performed on an IN_FILE.

The alternative approach would have been to associate a mode with a file when this file is assigned a value by an OPEN or CREATE operation. In that case each READ operation would have had to start with the check

```
if MODE(FILE) = OUTPUT then
   raise INVALID_MODE;
end if;
```

Note that this possibility of compile time checking of mode restrictions can be achieved at no extra complexity for the user because of the overloading facility. Primitives can be defined for different file types that have the same name, so that the number of distinct primitives to be remembered by the user is not increased.

Additional power is provided for the control of open and closed files. CREATE and OPEN should not be applied to files that are already open, for fear of destroying any path to an open file. A check on this can be performed by passing the file as an **in out** parameter to CREATE or OPEN, which can perform the test

```
if IS_OPEN(FILE) then
   raise FILE_OPEN_ERROR;
end if;
```

Clearly, this would lead to problems with uninitialized variables on the first CREATE or OPEN on a file, but the ability to provide a default initialization of record components allows us to solve this problem by a declaration of the form

```
generic(type ELEMENT_TYPE)
package INPUT_OUTPUT is
   restricted type IN_FILE     is private;
   restricted type OUT_FILE    is private;
   restricted type INOUT_FILE  is private;
   ...
private
   NO_FILE : constant INTEGER := 0;
   type BASIC_FILE is
      record
         FILE_INDEX : INTEGER := NO_FILE;
      end;
   type IN_FILE     is new BASIC_FILE;
   type OUT_FILE    is new BASIC_FILE;
   type IN_OUT_FILE is new BASIC_FILE;
end INPUT_OUTPUT;
```

Thus with the declarations

```
F  : IN_FILE;
G  : IN_FILE;
```

the two files are initialized and the creation check can be performed. Similarly since the files are passed as **in out** parameters to CLOSE and DELETE, these operations can reset the internal value to the value NO_FILE corresponding to a closed file. Note also that it may be essential for the package to ensure that file names are not lost. This could happen if the user were allowed to assign file variables such as F and G, but this is prohibited by the fact that IN_FILE is a restricted type.

Another important aspect of the language for the definition of input-output is the ability to parameterize a package and the enclosed entities (such as file types) by a generic clause. This facility permits the definition of as many file types as needed. For example the declarations

```
package INT_IO   is new INPUT_OUTPUT(INTEGER);
package REAL_IO  is new INPUT_OUTPUT(REAL);
```

define complete instantiations of the package INPUT_OUTPUT for the corresponding element types. Consequently the file variables INT_FILE and REAL_FILE in the following declarations refer to two distinct file types:

```
INT_FILE   : INT_IO.IN_FILE;
REAL_FILE  : REAL_IO.IN_FILE;
```

Another example of use of the generic facility is for the definition of input-output for enumeration types. The corresponding package has the form

```
generic( type ENUM_TYPE;
         function REP(X : ENUM_TYPE)  return STRING is   ENUM_TYPE'REP;
         function VAL(X : STRING)     return ENUM_TYPE is ENUM_TYPE'VAL)
package ENUM_IO is
  ...
  procedure PUT( FILE    : OUT_FILE;
                 ITEM    : ENUM_TYPE;
                 WIDTH   : INTEGER := 0);
  ...
end;
```

When instantiating such a package one can take advantage of the default parameters and specify for example

```
package COLOR_IO is new ENUM_IO (COLOR);
```

Within the package body, the body of the procedure PUT can be expressed in terms of the PUT function available for strings

```
procedure PUT(FILE: OUT_FILE; ITEM: ENUM_TYPE; WIDTH: INTEGER := 0) is
  S : STRING := REP(ITEM);
begin
  -- check column, etc.
  PUT(FILE, RIGHT_PAD(S, WIDTH));
end;
```

All read and write operations can be mapped onto a more primitive package dealing with bit strings. The basic read and write operations take as argument a file (for example of the type BASIC_FILE defined above), an address expressed as a number of storage units, and a size expressed in bits. As a result, a READ operation gets the appropriate number of bits from the designated file, and stores them starting at the indicated address. Similarly, WRITE copies the appropriate number of bits starting at the indicated address onto the designated file. The predefined attributes ADDRESS and SIZE can be used to express the typed form of READ and WRITE procedures in terms of the basic (untyped) form. For example WRITE could call a more basic procedure BASIC_WRITE as follows

```
BASIC_WRITE(BASIC_FILE(FILE), ITEM'ADDRESS, ITEM'SIZE);
```

Two limitations of this procedural approach to input-output should be mentioned here. First the language does not have any concept equivalent to the *straightening* of Algol 68. If straightening were provided, a procedure such as PUT (which is defined for the type INTEGER) would be automatically extended (by iteration) to arrays of integers. Similarly if PUT is defined for the types of the components of the record, it would also be defined for the record type itself. The second limitation concerns parameter lists. It is traditional to perform several output operations with a single order but this is not permitted by a strictly procedural form. Conceivably one could use another separator (say //) to permit multiple argument lists for a procedure:

```
PUT("X = " // X // "Y = " // Y);  -- not in Green
```

For simplicity neither straightening nor multiple parameter lists have been retained in Green. The choice of short identifiers reduces the inconvenience of the procedural form:

```
PUT("X = "); PUT(X); PUT("Y = "); PUT(Y);
```

In conclusion, we summarize the features of the language that are especially useful for defining the standard input-output package:

- generic packages
- explicit generic instantiation
- default parameters (both for procedures and for generic packages)
- overloading
- restricted types
- exceptions
- default record initialization

Additional features, such as the use clause and the inheritance of type properties have been used in the text input-output package.

Bibliography

[ANSI 66] ANSI (formerly USASI) X3.9 - 1966 (USA Standard Fortran).

[Ba 70] Balzer, R.M., Ports, A method for dynamic interprogram communication and job control. Rand Corporation (R605 ARPA) (1970).

[Ba 76] Barnes, J.G.P., RTL/2 design and philosophy, Heyden, London (1976).

[BFH 76] Bron, C., Fokkinga, M.M. and De Haas, A.C.M., A proposal for dealing with abnormal termination of programs, Twente University of Technology, Mem. Nr. 150, The Netherlands (Nov. 1976).

[BJ 78] Bert, D., Jacquet, P., Some Validation Problems with Parameterized Types and Generic Functions, Proc. 3rd International Symposium on Programming, Paris, Dunod (1978), pp. 279-292.

[BH 70] Brinch Hansen, P., The Nucleus of a Multiprogramming System, Comm. ACM, Vol. 13,4 (April 1970), pp. 238-241.

[BH 73] Brinch Hansen, P., Operating System Principles, Prentice Hall, N.J. 1973.

[BH 75] Brinch Hansen, P., The Programming Language Concurrent Pascal, IEEE Trans. Soft. Eng., 1,2(June 1975), pp. 199-207.

[BH 77] Brinch Hansen, P., Distributed processes, a concurrent programming concept, Computer Science Department, University of Southern California - Los Angeles (1977).

[BH 78] Brinch Hansen, P., Distributed processes: A Concurrent Programming Concept, Comm. ACM, Vol. 21, No. 11, November 1978, pp. 934-941.

[Br 75] Brosgol, Ben M., et al., CS4 Language Reference Manual and Operating System Interface, Report IR-130-2, Intermetrics, Inc., Cambridge, Mass., October 1975.

[Br 78] Brown, R.S., A Realistic Model of Floating Point Compilation, Numerical Software, Vol. 3 (ed., Rice), Academic Press, 1978.

[CH 71] Clark, B.L., Horning, J.J., The System Language for Project Sue, Sigplan Notices, Vol. 6,9, pp. 79-88 (Oct. 1971).

[CH 74] Campbell, R.H., Habermann, A.N., The Specification of process synchronization by path expressions, Lecture notes in computer sciences, Vol. 16, Springer (1974), pp. 89-102.

[Co 63] Conway, M.E., Design of a separable transition diagram compiler, Comm. ACM 6,7 (July 1963), pp. 396-408.

[DEC 74] Digital Equipment Corporation, BLISS-II Programmer's Manual, Maynard, Mass. (1974).

[DH 76] De Haas, A.C.M., Escape Clauses in Programming Languages, Twente University of Technology, The Netherlands (Sept. 1976).

[Di 68] Dijkstra, E.W., Cooperating Sequential Processes, in Programming languages, (ed. F. Genuys), Academic Press, London, 1968.

[Di 72] Dijkstra, E.W., Notes on Structured Programming, in Structured Programming, Academic Press, London, 1972.

[DoD 78] Department of Defense STEELMAN requirements for high order computer programming languages, June 1978.

[DNM 69] Dahl, O.J., Nygaard, K., Myhrhaug, B., The Simula 67 common base language, Pub S-22, Norwegian Computing Center, Oslo (1969).

[Fo 77] Fokkinga, M.M., Axiomatization of declarations and the formal treatment of an escape construct, Twente University of Technology, Mem. Nr. 176, The Netherlands (Sept. 1977).

[Fr 77] Francez, N., Another advantage of Keyword Notation for Parameter Communication with Subprograms, Comm. ACM 20,8 (Aug. 1977) pp. 604-605.

[GH 75] Gannon, J.D., Horning, J.J., Language design for programming reliability, IEEE Trans., Software Eng. SE-1,2 (June 1975) pp. 179-191.

[GK 77] Goos, G., Karstens, P., A Comparison of Modularization Facilities in Four Languages, IFIP Working Conference on the Production of Reliable Software, Novosibirsk 1977, North Holland.

[GMS 77] Geschke, C.M., Morris, J.H., Satterthwaite, E.H., Early Experience with Mesa, Comm. ACM 20,8 (Aug. 1977) pp. 540-553.

[Go 75] Goodenough, J.B., Exception Handling: Issues and a Proposed Notation, Comm. ACM 18,12 (Dec. 1975) pp. 683-696.

[GS 77] Geschke, C.M., Satterthwaite, E.H., Exception Handling in Mesa, XEROX PARC report, Palo Alto (1977).

[Gu 77] Guttag, J., Abstract Data Types and the Development of Data Structures, Comm. ACM 20,6 (June 1977)pp. 396-404.

[Ha 77] Hartmann, A.C., A concurrent Pascal Compiler for a mini computer, Lecture notes in Comp. Science, Vol. 50, Springer Verlag (1977).

[Hab 73] Habermann, A.N., Critical comments on the programming language Pascal, Acta Informatica, 3(1973), pp. 47-57.

[Har 76] Hardgrave, W.T., Positional versus keyword parameter communication in programming language, SIGPLAN Notices (ACM) 11,5 (May 1976), pp. 52-58.

[HF 76] Hague, S.J., and Ford, B., Portability, Prediction and Correction, Software Practice and Experience, Vol. 6 (1976), pp. 61-69.

[HLMR 74] Horning, J.J., Lauer, H.C., Melliar-Smith, P.M., Randell, B., A program structure for error definition and recovery, in Operating Systems, Gelenbe and Kaiser (Eds.), Springer Verlag, N.Y. (1974), pp. 171-187.

[Ho 74] Hoare, C.A.R., Monitors: An Operating System Structuring Concept, Comm. ACM, Vol. 17, No. 10, Oct. 1974, pp. 549-557.

[Ho 78] Hoare, C.A.R., Communicating sequential processes, Comm. ACM, Vol. 21, No. 8, August 1978, pp. 666-677.

[HW 79] Holt, R. C., Wortman, D., B., A model for implementing Euclid modules and type templates (to be published) 1979.

[IF 77] Ichbiah, J.D., Ferran, G., Separate Definition and Compilation in LIS and its Implementation, Cornell Symposium on the Design of High Order Languages, in Lecture Notes in Computer Science, Springer Verlag, N.Y.(1977).

[IRHC 74] Ichbiah, J.D., Rissen, J.P., H' eliard, J.C., Cousot, P., The System Implementation Language LIS, Reference Manual, CII-HB Technical Report 4549 E/EN, Dec. 1974, CII-HB, Louveciennes, France.

[Ka 74] Kahn, G., The semantics of a simple language for parallel programming, Proc. IFIP Congress 74, North Holland (1974).

[Le 77] Levin, R., Program Structures for Exceptional Condition Handling (PhD Thesis), Dept. of Computer Science, Carnegie Mellon University (June 1977).

[LHLMP 76] Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G. and Popek, G.J., Report on the *Programming Language Euclid, Xerox report*, Palo Alto (Aug. 1976).

[Li 74] Liskov, B. A Note on CLU, Computation Structures Group Memo 112, MIT Project MAC, Cambridge, Mass., November 1974.

[LMS 74] Lampson, B.W., Mitchell, J.G., Satterthwaite, E.H., On the transfer of control between contexts, Lecture notes in Computer Science, Vol. 19, Robinet (Ed.) Springer Verlag, N.Y. (1974) pp. 181-203.

[LS 79] Lauer, H.C., Satterthwaite, E.H., The impact of Mesa on system design (to be published) 1979.

[LSA 77] Liskov, B., Snyder, A., Atkinson, R., Abstraction mechanisms in CLU, Comm. ACM 20,8 (Aug. 1977), pp. 564-576.

[Or 50] Orwell, G., "1984", Harcourt, Brace and Co., N.Y. (July 1950).

[Pa 71] Parnas, D.L., Information distribution aspects of design methodology, Information Processing 71, North Holland Pub. Co., Amsterdam (1971), pp. 339-344.

[Pe 66] Perstein, M.H., The JOVIAL (J3) Grammar and Lexicon, SDC technical report TM-555 (1966).

[PW 74] Presser, L., White, J., Making Global Variables Beneficial, Proceedings IFIP (Aug. 1974).

[PW 76] Parnas, D.L., Wurges, H., Response to Undesired Events in Software Systems, Proc. 2nd Int'l. Conf. on Software Engineering, (October 1976).

[R 74] Rain, M., Mary Programmers Reference Manual, R Unit, Trondheim (1974).

[Ra 75] Randell, B., System Structure for Fault Tolerance, Proc. Int'l. Conf. on Reliable Software, Los Angeles (1975).

[Ro 70] Ross, D.T., Uniform referents: an essential property for a software engineering language, in Software Engineering (J.TOU, Ed.), Vol. 1, Academic Press (1970), pp. 91-101.

[SHW 78] Shaw, M., Hilfinger, P., Wulf, W.A., TARTAN - Language Design for the Ironman Requirements: Reference Manual, SIGPLAN Notices, Vol. 13,9 (September 1978), pp. 36-58.

[SW 74] Scowen, R.S., Wichmann, B.A., The definition of comments in Programming Languages, Software-Practice and Experience, Vol. 4,2 (April 1974), pp. 181-188.

[VH 75] Von Henke, F.W., On Generating Programs from Data Types: An Approach to Automatic Programming. Proc. Int'l. Symposium on Proving and Improving Programings (Arc-et-Senans) IRIA (1975), pp. 57-70.

[VW 75] Van Wijngaarden, A., et al., Revised Report on the algorithmic Language Algol 68, Acta Informatica, Vol. 5, Fasc. 1-3 (1975).

[Wa 72] Walden D.C., A System for Interprocess Communication in a Resource Sharing Computer Network, Comm. ACM 15,4 (April 1972) pp. 221-230.

[We 74] Wegbreit, B., The Treatment of Data Types in EL1, Comm. ACM, Vol. 17,5 (May, 1974), pp. 251-264.

[We 78] Welsh, J., Economic Range checks in Pascal, Software - Practice and Experience, Vol. 8, 1 (Jan 1978) pp. 85-98.

[Wi 71] Wirth, N., The programming language Pascal, Acta Informatica, Vol. 1, No. 1 (1971), pp. 35-63, Springer Verlag.

[Wi 71b] Wirth, N., Program development by stepwise refinement, Comm. ACM, 71, 14, p. 221.

[Wi 76] Wirth, N., Modula, A language for modular programming, Technische Hochschule Zurich, report 18 (March 1976).

[WLS 76] Wulf, W.A., London, R.L., and Shaw, M., Abstraction and verification in Alphard: Introduction to language and metholodology, USC Inform. Sci. tech. Report, University of Southern California, Los Angeles (1976).

[Wo 72] Woodger, M., Levels of Languages in High Level Languages, INFOTECH, State of the Art Report, No. 7, pp. 201-215.

[WSH 77] Welsh, J., Sneeringer, M.J., Hoare, C.A.R., Ambiguities and Insecurities in Pascal, Software-Practice and Experience, Vol. 7,6 (November 1977), pp. 685-696.

[WWG 70] Woodward, P.M., Wetherall, P.R., and Gorman, B., Official Definition of CORAL 66, Her Majesty's Stationery Office (1970).

[Za 74] Zahn, C.T., A control Statement for natural top-down structured programming, in lecture Notes in Computer Science, No. 19, Robinet(Ed.), Springer Verlag, N.Y. (1974), pp. 170-180.

Index

References in parentheses denote the main entry in the Reference Manual.